Broadview
www.broadview.com.cn



NiJili's Series
倪继利作品系列

凝聚研发经验
具备专业深度

倪继利 编著

Qt及Linux操作系统窗口设计

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

前　言

背景知识

Linux 操作系统已被广泛地使用到 PC 和各种嵌入式设备。笔者在《Linux 内核分析及编程》（电子工业出版社 2005 年 9 月出版）中分析了内核的各种机制，本书作为它的配套书，主要讲述 Linux 操作系统的窗口系统设计。

目前主流的 Linux 窗口系统有：PC 上使用的基于开放源代码的 Gnome 和 KDE、嵌入式设备上使用的开放源代码 Opie 和 Gpe 以及一些商业化的版本（如 Motorola 的 PDA 手机的窗口系统）。

GPE（Gpe Palmtop Environment） 窗口系统最类似于桌面领域中的 GNOME，它们都基于 GTK，即一种基于 C 语言的 GUI 界面开发工具。

Opie（Open Palmtop Integrated Environment）窗口系统类似于桌面领域中的 KDE，Opie 基于 Qtopia，Qtopia 是基于 Qt/Embedded 的简化窗口系统。Opie 在 Qtopia 的窗口系统机制上扩展了类库及应用程序，从而更加实用化，可以在多款 PDA 及 PDA 手机上完美运行。KDE 窗口系统基于 Qt/X11，Qt/X11 又基于 X Window 系统。Qt/X11 与 Qt/Embedded 具有相同接口的类库，对于应用程序编程者来说，可以使用相同的类进行开发。 Qt 是基于面向对象的 C++的 GUI 界面开发工具，另外还提供了 KDevelop 和 QDesigner IDE 图形界面开发环境。

由于 Qt 具有面向对象、技术成熟等优点，并且目前高端嵌入式设备生产商几乎都选择 Qt 作为开发工具，因此本书选择 Qt 作为界面开发工具，选择 KDE 作为 PC 桌面环境，选择 Qtopia 嵌入式设备窗口系统进行分析介绍。

操作系统的窗口系统内容很广泛，对于 PC，KDE 桌面环境包括 KOffice 办公软件、网页/文件浏览器和系统工具等应用程序，它所依赖的 X Window 系统包括多种协议及 API 库；对于嵌入式设备，Opie 包括了许多应用程序， Qt/Embedded 包括了许多类库。这么多的内容在一本书中要想全面涉及是不可能的，本书主要针对专业开发所需要的深度，有重点地进行讲述。

本书特色与重点

本书主要讲述如何在 Linux 内核上建立窗口系统。

对于 PC 窗口系统，本书重点介绍 KDE 窗口系统如何在启动时从 Linux 内核上建立起来，以及输入设备及图形显示卡硬件如何与窗口界面中的控件进行交互；对于嵌入式设备窗口系统，介绍了 Qtopia 的应用程序管理器如何从 Linux 内核上启动运行，以及输入设备及图形显示卡硬件如何通过 Qt/Embedded 客户/服务器与窗口界面中的控件进行交互。由于 KDE 与 Qtopia 的窗口系统运行机制有许多相似之处，因此，本书重点讲述了 Qtopia 的应用程序管理器以及 KDE 的核心技术。笔者认为只要精通了 Qtopia 的编程，知道了 KDE 的核心技术以及与 Qtopia 编程的差别，并且仔细阅读 KDE 类库的说明文档，也能编写 KDE 应用程序。

在本书中，笔者几乎没有对 KDE 及 Qt 的类库进行说明。因为它们的 API 类库说明文档已说明得很详细，还可以参照源代码。通过对照阅读，读者就会明白各种类的用法。

关于作者

笔者从清华大学电子工程系读研究生起，就开始从事 Linux 内核编程工作，之后又一直在外国著名公司从事 Linux 内核编程工作；除了内核编程工作外，还从事过嵌入式设备的文件管理器及应用程序管理器等的开发工作。

读者对象及阅读方法

本书的读者对象为从事或准备从事 Linux 嵌入式设备开发的人员。为阅读本书，读者需具备 C++ 面向对象编程方面的知识。

建议读者在阅读本书时在 PC Linux 操作系统下编译运行 X 86 平台的 Qtopia，然后对照所运行的 Qtopia 窗口系统和 Qtopia 源代码来阅读，效果会更好。

写作目的

本书的写作目的是让读者对 Linux 窗口的建立过程，以及应用程序从窗口系统启动、文档的打开过程有个清晰的了解。本书中介绍的窗口的分层设计、接口标准化、菜单的通用化，以及应用程序查询硬件信息等设计方法，开发者有较高的参考价值。同时，本书还分析了 Qt 应用程序的编译过程，即 qmake 命令如何将项目文件转换成 Makefile 、依赖关系如何传递等，这些内容对开发者查找编译错误可提供清晰的思路。

结构安排

笔者认为，对于应用程序编程者来说，应该重点掌握类的继承关系及对象管理链表、事件、进程间通信、应用程序与桌面环境的交互等。因此，本书重点介绍了这些内容，而舍去了对类库及利用类库编写应用程序方面的介绍。

本书介绍了 Qtopia，而没有介绍 Opie。这是因为 Opie 使用了与 Qtopia 一样的机制，Opie 只是扩展了类库与应用程序。更重要的是 Qtopia 由 Trolltech 公司维护,其提供了各种详细的文档。而且 Qtopia 代码量少，结构清晰，便于理解。如果读者理解了 Qtopia，那么再看 Opie 代码就会很容易理解。

Opie 中的代码更接近实际的产品，甚至 Opie 中的应用程序比许多产品化的应用程序写得还要完善，只要对它进行简单的修改便可用到产品中。

笔者长期从事 Linux 内核程序的开发，也经常要开发应用程序界面，比如使用 Qt开发嵌入式设备的应用程序界面。有时，笔者还借助于应用程序界面与内核模块共同完成某些功能，如病毒防火墙。由于版权问题，笔者不可能将工作中开发的代码用做书中的源代码。但笔者在本书中尽量选择最贴近产品开发的开放源代码进行讲解，力求对读者的实际开发工作有帮助。

章节简介

第 1 章 “X Window ”阐述了 X 的体系分层结构、显卡驱动程序、 X协议，说明了建立在 X上的各种编程方法，还分析了 X 客户端应用程序 Xlib 函数如何发出请求以及 X服务器的工作原理及源代码。

第 2 章 “Qt 编程核心技术”介绍了 Qt 对象模型、国际化方法、元对象及代码生成、进程间通信、窗口部件的基类、模板库和集合类、 Qt 线程、鼠标拖放、键盘焦点、会话管理，以及调试等方面的技术。

第 3 章 “KDE 窗口系统”介绍了 KDE 的 MIME 类型处理、服务、KIO 框架、组件框架和国际化等，还分析了 KDE 的启动过程及 kicker 应用程序。

第 4 章 “Qt/Embedded 客户/服务器”讲述了将应用程序下载到嵌入式设备运行的方法，介绍了各种嵌入式 GUI 窗口系统，阐述了 Qt/Embedded 的客户/服务器模型工作过程。本章还分析了设备输入事件在客户与服务器之间的传递机制、应用程序窗口的显示原理，并说明了运行 Qt/Embedded 程序的方法。

第 5 章 “Qtopia 核心技术”介绍了 Qtopia 开放源代码的核心技术及类库，其中着重分析了插件的装载过程、应用程序快速启动器的机制、类 MimeType ，以及编写输入法的方法等。

第 6 章 “Qtopia 服务器”主要说明组成 Qtopia 服务器的各个类的功能，介绍了 Qtopia 服务器的启动过程，分析了 Qtopia 服务器如何启动应用程序及插件，还分析了电源监控、外观设置、应用程序安装与卸载以及 AppServices 等与系统管理相关的小应用程序。

第 7 章 “Qtopia 的 sysinfo 应用程序分析”详细分析了 sysinfo 应用程序的源代码。

第 8 章 “Qtopia 编译及系统集成”说明了 Qtopia 的定制与集成，阐述了 .pro 项目文件的语法。并且详细分析了由 .pro 项目文件生成 Makefile 文件的过程以及描述了 Qtopia 的打包、安装和调试方法。

本书用到的软件版本

Qtopia 2.1.1
Qt/Embedded 3.4
Qt/X11 3.4
KDE 3.5
X Window 的 x11R682

致谢

由于有上一本书与电子工业出版社博文视点公司成功合作的愉快经历，所以笔者继续将此书信任地交给该公司的朋友们编辑出版。郭立女士及她的同事对本书的关注和用心使笔者非常感激，感谢他们为我的上一本书及本书所做的大量工作。

建议与问题

笔者的上一本书《Linux 内核分析及编程》出版后，许多读者通过网络发表了自己的评论和意见。为了方便读者发表自己的见解，笔者建立了自己的工作博客，网址如下：

http://nijili.blogcn.com

欢迎读者和专家访问指教，您可以提出自己的建议，指出书中的错误，还可以与网友一起分享和讨论阅读本书的收获以及有利于本书进一步改进的所有问题。

倪继利

2005 年 12 月 28 日于北京

第 1 章　X Window

1984 年，麻省理工学院（MIT）电脑科学研究室开始开发 X Window。1986 年，MIT 和数字设备公司（DEC）对 X Window 协议进行重新设计，推出了 X Window 第 11 版，即 X11。

X11 协议定义一个系统所必须具备的功能，任何系统若能满足此协议并符合 X 协会的其他规范，便可称为 X。

XFree86 是基于 Intel PC 平台的 X 系统，其含义是“提供基于 Intel 的 PC 平台，自由的 X 服务”。

X Window 是高度可配置的，它只定义了一系列最基本的显示功能调用，而没有规定图形窗口的样式。图形窗口的样式由程序员开发的窗口管理程序完成，从而使得 X Window 的风格多种多样，用户可根据自己的喜好定制图形界面。

本章阐述了 X 的体系分层结构、显卡驱动程序、X 协议，说明了建立在 X 上的各种编程方法，还分析了 X 客户端应用程序 Xlib 函数如何发出请求，以及 X 服务器的工作原理及源代码。

1.1　X Window 的体系结构

X Window 的运行基于一种客户 /服务器（Client/Server）模式，它由 X Server、X Clients 和通信通道 3 个部分组成，X Server 和 X Clients 通过 X 协议在网络上通信完成应用任务。

X Server 是控制输入输出的程序，它和底层硬件直接通信控制实际的显示器、鼠标及键盘的软件，它只在接收 X Client 程序的请求后完成建立窗口和绘制图形等工作。

X Client 是基于 X Server 的客户程序，作为使用系统的视窗功能的程序，它请求 X Server 在指定的窗口执行各种操作。它不负责显示，只是给 X Server 发送一个请求，由服务器完成操作。

X 协议是 X Server 和 X Client 之间沟通的语言，Xlib 库封装了可完成这种通信功能的 API，可以使用这些 API 开发 X Client 程序。

X Window 运行分层如图 1-1 所示。

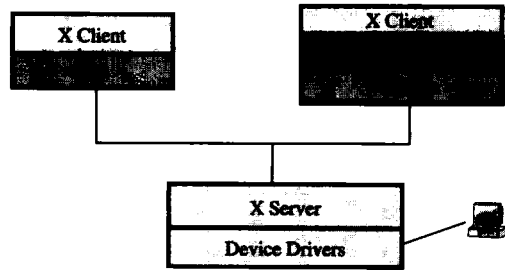


图 1-1　X Window 运行分层

其中最底层的是 X Server，其上层是 X 网络协议，该层使远程运行 X Window 成为可能。再上层是称为“ Xlib”的底层函数接口，它介于网络和基础系统与较高层的程序之间，应用程序的实现通过调用这一层的函数实现。最顶层是管理窗口的窗口管理器，即 WM（Window Manager），如 AfterStep、Blackbox、Enlightenment、ctwm、ftwm、sawfish、twm 和 Window Maker 等，这些管理器中的每一个

都提供了一个不同的界面。另外，KDE 和 GNOME 桌面环境（Desktop Environments）都有自己的视窗管理器与桌面集成，每一个视窗管理器也有一个不同的配置机制。

X Window 的客户机/服务器模式有如下的优点。

（1）客户程序可以在远程计算机上执行计算任务，而使用的 X 服务器仅负责复杂的图形显示，从而充分发挥 X 服务器在显示上的优势。

（2）只有 X 服务器与硬件打交道，所有的客户程序都与硬件无关，从而很容易在不同的平台上移植。

（3）使用不同的视窗管理器会使得 X Window 的外观看起来截然不同。

1.1.1 X Server

X Server 是控制显示器和输入设备（键盘和鼠标）及一些 X 资源的软件，与硬件设备相关，处理 X Client 所传递的 X 事件。X 资源包括色彩和字形等，X 事件包括键盘的输入、鼠标移动及窗口大小改变，等。

每一套显示设备只对应一个惟一的 Server，X Server 通常根据其管理的显卡类型来命名，如 XF86_SVGA、XF86 VGA16 和 XF86 MONO 等。

X Server 的实现可以是单线程或多线程的，现有的 X Server 包括 Workstation、X Terminal、PC X Server 和 Web Browser。

X11R5 增加了一项“字库服务器”（font server）功能，其基本概念是将字库分布存储在网络中的机器上，每个 X Server 都可以通过网络取得所需要的字型。X Server 可以没有自己的字库，在一个网络中一套字库只须存储一份，如图 1-2 所示。

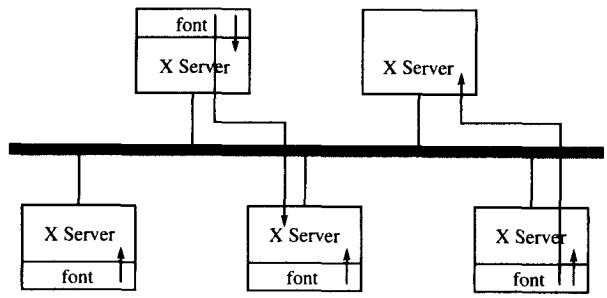


图 1-2 X font server

X font server 的结构非常灵活，一个 X Server 可以同时读取多处的字库，一个 X font server 也可以同时为多个 X Server 服务。

X font server 还可以是层次体系的，即一个 X Server 可以通过一个 font server 访问其他的 font server，如图 1-3 所示。

1.1.2 X Client

X Client 是使用系统视窗功能的一些应用程序，如窗口管理器（Windows Manager，WM）。X Client 与硬件设备无关，无法直接影响视窗，而只能送一个请求（request）给 Server，由 Server 来完成其请求。

典型的 X 客户程序有窗口管理器和桌面系统，窗口管理器决定窗口外观的客户进程，可改变窗口的大小或位置、将窗口缩成图标，并重新安排窗口在堆栈中的位置等。Linux 支持多种窗口管理器，如 fvwm 和 Kdm 等。

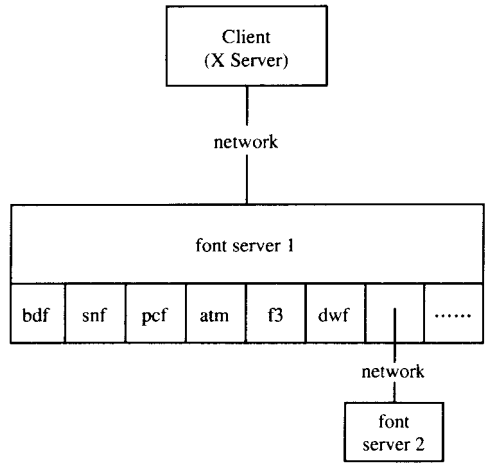


图 1-3　X font server 的层次体系

桌面系统是一个客户进程，它控制桌面上应用程序图标、“开始”菜单及其内容，以及鼠标在桌面上进行键击和拖动操作所产生的效果等。桌面系统集成了窗口管理器和其他工具，目前 Linux 系统上主要的桌面系统有 KDE和 GNOME。KDE采用 Kdm 作为窗口管理器。

用户可以通过系统提供的程序或第三方软件使用 Client 程序，或者为了某种特殊应用而编写自己的 Client 程序。

1.1.3　X Protocol

X Protocol 定义了应用及其显示之间的客户 /服务器关系，通过这个协议，应用与其显示被分离开来。更进一步，X 协议被定义为两层，即设备相关和设备无关。X Client与 X Server 之间的通信是异步的双向协议，任何提供字节流通信的方式都可以使用。可以是 IPC，也可以是 TCP/IP。X Protocol 隐藏了操作系统和硬件的特殊性，这样大大简化了应用的开发和 X Window 系统的可移植性。

X Server 和 X Client 通信有如下两种基本操作模式。

（1）X Server和 X Client 在同一台机器上运行，它们可以使用机器上任何可用的通信方法。在这种情况下，X 可以同其他传统的视窗系统一样高效工作。

（2）X Client 和 X Server 在不同机器上运行，二者通信就必须遵守网络协议进行，最常用的协议为 TCP/IP。通过测试 TCP/IP，可以看到 X Server 在 TCP 6000 端口上侦听。

例如如果应用程序在远程计算机上运行，输入与显示在本地计算机上运行，那么本地计算机是 X Server，远程计算机是 Client，可使用类似如下命令：

```
rsh remote_pc run_program -display local_pc:0
```

这个命令的含义是在名为 remote_pc 的远程计算机上运行程序 run_program ，结果显示在名为 local_pc 的本地计算机的第 0 号显示器上。

X Server 与 X Client 间的交互如图 1-4 所示，通信过程如下。

（1）用户通过鼠标键盘对 X Server 下达操作命令。

（2）X Server利用 Event 给 X Client 传递用户操作信息。

（3）X Client 执行处理。

（4）X Client 利用 Request 传回所要显示的结果。

（5）X Server 将结果显示在屏幕上。

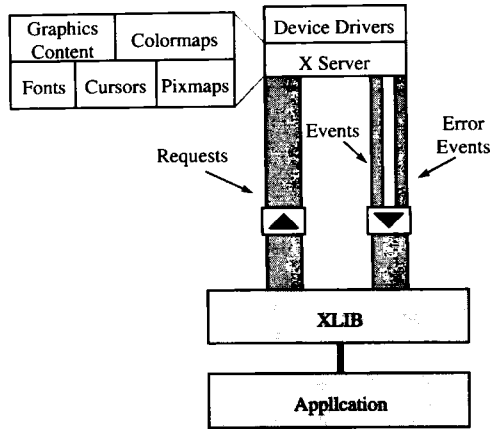


图 1-4 X Server 与 X Client 间的交互

X 协议具有以下客户机和服务器之间的主要通信信息。

（1）请求

X Client 向 X Server 发出服务请求，如创建窗口等。为了提高速度， X Client 通常不等待响应。请求保留给可靠的网络层进行传送。X 协议的请求是 4 个字节的倍数。

（2）答复

答复是 X Server 是对某个的 X Client 请求的响应，但并不是所有的请求都需要响应。 X 协议的答复是 4字节的整数倍，最小为 32 个字节。

（3）事件

X Server 转发事件给 X Client，这些事件包括键盘或鼠标输入等。 X 事件为 32 个字节。

（4）差错

X Server 应 X Client 请求报告差错。X 差错大小与事件相同，它们被发送到 X Client 的差错处理程序。X 差错为 32 个字节。

1.2 窗口管理器

X Window 的设计目的是支持多种不同的用户界面，它提供了一般性的架构。即由不同的窗口管理器或桌面环境提供具体的窗口样式，如可以选择 Window Maker 窗口管理器或桌面环境 KDE（K Desktop Environment），它们都建立在 X Window 之上。

窗口管理器是控制窗口属性的一个 Client 程序，它管理窗口的外观形式、桌面菜单、图标、虚拟桌面，以及按钮等显示的样式。作为系统的一部分，它控制屏幕最上层的窗口，其功能包括改变窗口大小或位置、将窗口系统叠放或排列平放或变为一个图标等。用户可以使用不同的窗口管理器，屏幕显示的桌面将因窗口管理器的不同而不同。

桌面环境是包括窗口管理器及各类整合工具和应用程序的集合， Linux 最常用的桌面环境有 GNOME 和 KDE。GNOME 是以 GTK+作为 GUI 图形开发的工具包，包括许多小工具和大量的应用软件。

常见的窗口管理器说明如下。

（1）Twm

Twm 全称为“标签窗口管理器”（Tab Window Manager），作为一种极为简单的 X Window 窗口管理器，它不使用任何图标。用户需要点击背景，打开菜单项来完成工作。对于机器内存较小的用户，

Twm 窗口管理器是其首选。

（2）Window Maker

Window Maker 是一个优秀的窗口管理器，除了提供常用的窗口管理器功能之外，还提供支持 XPM、PNG、JPEG、TIFF、GIF 及 PPM 图标，以及多国语言等。其特色是在不重新启动 X 的状态下，可以修改 Window Maker 的菜单、界面颜色，以及字体。

（3）Fvwm

Fvwm 是一个仅拥有最基本框架的窗口管理器，是在最早的 Twm 窗口管理器基础上开发的。它没有提供华丽的界面和强大的功能，比较适合于性能比较差的计算机。

（4）mwm

mwm（Motif Window Manager）遵循 OSF（Open Source Foundation，开放源代码基金会）制定的 Motif 窗口管理标准。Motif 付费后才能使用。

（5）Sawfish

Sawfish 是使用 Lisp脚本语言开发的可扩展窗口管理器，用户可以通过修改个人配置文件 .sawfishrc 中的 Lisp 代码，或者通过整合的用户定制系统来修改配置信息、更改组件并控制用户界面。

Sawfish 提供了菜单、交互式的窗口移动和改变大小、虚拟桌面、框架主题定义，以及其他标准窗口管理器类似的功能，是 GNOME 默认的窗口管理器。

1.3 X Window 启动过程

启动 X Window 的方法，第一是在/etc/inittab 文件中，把系统的默认运行级别定为 5，重新开机启动系统即可进入 X Window；二是用 startx 脚本手动启动。

startx 主要设置 client 和 server 脚本文件所在的位置并处理相关参数，最后交给 xinit 处理。xinit 根据 startx 传递的参数启动 X Server，成功后根据 xinitrc 启动 X Client。在启动 X Client 时根据脚本 X Client 的配置，可以启动窗口管理器，如 Twm 等，也可以启动 KDE 桌面系统。

当需要启动窗口系统时，在终端窗口输入 startx 命令就可以启动窗口系统，下面是 startx 脚本源代码分析：

```
#!/bin/sh
userclientrc=$HOME/.xinitrc      #  用户定义的 client 脚本文件
userserverrc=$HOME/.xserverrc  #      用户定义的 server  脚本文件
sysclientrc=/usr/X11R6/lib/X11/xinit/xinitrc   #系统定义的 client 脚本文件
sysserverrc= /usr/X11R6/lib/X11/xinit/xserverrc  #系统定义的 server 脚本文件
defaultclient=/usr/X11R6/bin/xterm      #默认的 client 运行程序
defaultserver=/usr/X11R6/bin/X       #    默认的 server 运行程序
defaultclientargs=""      # 默认的 client 参数变量
defaultserverargs=""       #默认的 server 参数变量
clientargs=""           #client 参数变量
serverargs=""          #server 参数变量

#选择 client 脚本文件
if[-f $userclientrc ]; then
   defaultclientargs=$userclientrc
elif[-f $sysclientrc ]; then
   defaultclientargs=$sysclientrc
fi

#选择 sever 脚本文件
if[-f $userserverrc ]; then
   defaultserverargs=$userserverrc
elif[-f $sysserverrc ]; then
   defaultserverargs=$sysserverrc
```

```
fi

# 下面循环处理 client 和 server 的参数
whoseargs="client"
while[x"$1" != x ]; do  #  运行 startx 所带第 1 个参数为空，退出循环
      case "$1" in
     # 需要阻止 cpp 把 /*" 当作 个 C注释
      #如果 $1 是/*或者./*形式(xinit 程序要求其参数里的 client程序和 server程序必须以 /或 ./开头 否则会被视为 client 程序和
server 程序的参数，见 man xinit)
      /''*|\./''*)
         if["$whoseargs" = "client" ]; then  #如果当前是在处理 client 的参数
            if[x"$clientargs" = x ]; then  # 如果 clientargs 为空，则赋值$1 给 client 变量
               client="$1"
            else
               #否则 clientargs 赋值为 $clientargs $1, $1 是 startx 命令行 options 参数
               clientargs="$clientargs $1"
            fi
         else  # 当前在处理 server 的参数，
            if[x"$serverargs" = x ]; then
              server="$1"
            else
              serverargs="$serverargs $1"
            fi
         fi
      ;;

      #如果 $1 为---，则表示开始处理 server 的参数， --为 client 和 server 参数的分界
      --)
      whoseargs="server"

      #为默认值时
      *)

      if["$whoseargs" = "client" ]; then # 处理给 client 程序的参数
         clientargs="$clientargs $1"
      else #处理给 server程序的参数

      #display 屏幕编号必须为第 1 个给 server程序的参数，以 :x 的形式(x 为数字)
         if[x"$serverargs" = x ] && expr "$1" : ':[0-9][0-9]*$' > /dev/null 2>&1; then
            display="$1"
         else #处理屏幕编号以外的参数
            serverargs="$serverargs $1"
         fi
      fi
      ;;
      esac
      shift  #所有参数左移次
      done

      # 处理 client 参数
      if[x"$client" = x ]; then  # 如果 client 程序为空
         #如果没有 client 参数，使用 rc 文件代替
         if[x"$clientargs" = x ]  ; then  #clientargs  为空
            client="$defaultclientargs"
         else
            client=$defaultclient     # 使用默认的 client 程序
         fi
      fi

      # 处理 server参数
      if[x"$server" = x ]; then
      # 如果没有 server 参数或 display，使用 rc 文件代替
         if[x"$serverargs" = x -a x"$display" = x ]     ; then
            server="$defaultserverargs"
         else
```

```
        server=$defaultserver
    fi
fi

#……省略授权代码若干
xinit $client $clientargs -- $server $display $serverargs  #把处理过的参数交由 xinit 程序处理
```

从上面代码可以看出，startx 为了查找 client 参数，先后查找$HOME/.xinitrc 和/etc/X11/xinit/xinitrc；为了查找 server 参数，先后查找 $HOME/.xserverrc 和/etc/X11/xinit/xserverrc。

该 startx 命令启动 X 时，实际上等价于命令 xinit $HOME/.xinitrc -- $HOME/.xserverrc ，或 xinit /usr/X11R6/lib/X11/xinit/xinitrc -- /usr/X11R6/lib/X11/xinit/xserver。

xinit 根据 startx 传递的参数启动 X Server，成功后根据 xinitrc 启动 X Client。

在单台计算机中，startx 启动过程如下。

（1）startx 查找 client 和 server 脚本的位置，得到相应参数并调用 xinit。

（2）xinit 根据传递的参数启动 X Server，成功后呼叫 X Client。xinit 先启动 X Server。X Server 建立自己的根窗口，并且一直监控着键盘及鼠标。如果没有 Client 程序需要被告知键盘或鼠标事件，键盘或鼠标的输入传入 X Server 后被放弃。

（3）根据 xinitrc 设置相关资源，启动窗口管理器，输入法和其他应用程序等 X Client 程序。如在/etc/X11/Xclients 脚本中，有如下语句运行 Twm 窗口管理器：

```
exec /usr/X11R6/bin/twm
```

对于内存较大的机器，用户可以使用 KDE 或者 GNOME。在 clients 脚本文件中使用下面的命令行运行 KDE 桌面系统：

```
exec /usr/X11R6/bin/startkde
```

1.4 XFree86 配置文件分析

X Window 系统的程序和字体大多放在/usr/X11R6 目录中，配置文件放在/etc/X11 目录中，初始化程序（如 xfs 字体服务器）放在系统初始化目录/etc/init.d 目录中。XFree86 系统目录说明如表 1-1 所示。

表 1-1 XFree86 系统目录说明

目　录	描　述
/etc/X11	X 配置文件
/usr/X11R6/bin	X 运行程序
/usr/X11R6/include	X11 程序运行所需的头文件和图形
/usr/X11R6/lib	X 程序共享库
/usr/X11R6/lib/X11	配置文件、应用程序资源文件文档、字体及国际化文件
/usr/X11R6/lib/modules	XFree86 模块（驱动程序、扩展文件和字体文件等）
/usr/X11R6/man	X man 手册
/usr/X11R6/share	各种 X 客户程序共享的资源文件

单机上运行的 XFree86 字体目录主要在/usr/X11R6/lib/X11/fonts 目录中，下面几个工具用于管理字体。

（1）mkfontdir：用来在字体目录下创建字体名数据库。

（2）xlsfonts：用来列出已安装字体。

（3）xfontsel：用来显示从字体集选中的各种字体。

（4）xfd：用来显示在 X 字体中的字符。

（5）xset –q：用来查看字体的路径。

在 fonts 目录下列有已安装的各种字体目录，如 100dpi/、75dpi/和 misc/等。每个目录中都有一个 fonts.dir 和一个 fonts.alias 文件，fonts.dir 文件是字体列表数据库，X 和 xfs 字体服务器用这个文件查找字体；fonts.alias 文件包含名字和别名，可以利用这个文件为已有的字体创建新名字。 encodings 目录中的文件是 Type1（也称为 “TrueType”）的字体编码，这些信息用来在显示器上绘制可缩放字体。

XFree86 还可使用远程 XFree86 服务器上的字体，从而减少本地的字体存储空间，该功能要通过配置字体服务器 xfs实现。为利用远程服务器上提供的字体，可用 xset +fp unix/remote.org:7100 命令把远程 X 服务器提供的字体增加到当前显示字体的路径中。

在 Redhat 各发行版本中提供了设置 X Window 的工具软件，如 Xconfigurator、XF86Setup 和 XF86Config，等，它们提供了修改 X Window 配置文件 XF86Config 的简便方法。XF86Config 保存在 /usr/X11R6/lib/X11/ 目录中，在 /etc 或者 /etc/X11 目录中有其一个符号连接，它是一个文本文件。

XF86Config 文件分为 File、Modules、InputDevice、Monitor、Device 和 Screen 等几个部分。XF86Config 文件的示例及注释如下：

```
# redhat-config-xfree86 创建的 XFree86 4 配置

#Server 使用的配置
Section "ServerLayout"
     Identifier    "Default Layout"
     Screen     0  "Screen0" 0 0
     InputDevice    "Mouse0" "CorePointer"
     InputDevice    "Keyboard0" "CoreKeyboard"
     InputDevice    "DevInputMice" "AlwaysCore"
EndSection

#这小节指出文件路径
Section "Files"
    #RGB 数据库路径，不包括扩展名，仅用于记录颜色值
     RgbPath      "/usr/X11R6/lib/X11/rgb"
    #使用独立于 X Server 的字体服务器产生字体，服务器为 unix,端口为 7100
     FontPath    "unix/:7100"
EndSection

#服务器加载指定的模块
Section "Module"
     Load  "dbe"
     Load  "extmod"
     Load  "fbdevhw"
     Load  "glx"
     Load  "record"
     Load  "freetype"
     Load  "type1"
EndSection

#定义键盘输入设备
Section "InputDevice"
     Identifier  "Keyboard0"      #输入设备 ID
     Driver      "keyboard"      #驱动程序
     Option       "XkbRules"  "xfree86"    # “协议”   “键盘类型名”
     Option       "XkbModel"  "pc105"
     Option       "XkbLayout"  "us"
EndSection

# 鼠标输入设备
Section "InputDevice"
```

```
    Identifier  "Mouse0"
    Driver      "mouse"
    Option        "Protocol" "IMPS/2"
    Option        "Device" "/dev/psaux"
    Option        "ZAxisMapping" "4 5"
    Option        "Emulate3Buttons" "no"
EndSection

Section "InputDevice"

# 如果正常的 CorePointer 鼠标不是 USB 鼠标，那么，这个输入设备能以 AlwaysCore 模式使用，允许用户同时使用 USB 鼠标
    Identifier  "DevInputMice"
    Driver      "mouse"
    Option        "Protocol" "IMPS/2"
    Option        "Device" "/dev/input/mice"
    Option        "ZAxisMapping" "4 5"
    Option        "Emulate3Buttons" "no"
EndSection

#显示器设置
Section "Monitor"
    Identifier   "Monitor0"     #设备 ID
    VendorName   "Monitor Vendor"
    ModelName    "Dell M782"
    DisplaySize  320    240
    HorizSync    30.0-85.0     #水平扫描频率，以 kHz 为单位
    VertRefresh  50.0-160.0    #垂直扫描频率，以 Hz 为单位
    Option       "dpms"
EndSection

#显卡数据
Section "Device"
    Identifier  "Videocard0"
    Driver      "i810"
    VendorName  "Videocard vendor"
    BoardName   "Intel 845"
EndSection

#X Window 的显示模式
Section "Screen"
    Identifier "Screen0"
    Device     "Videocard0"
    Monitor    "Monitor0"
    DefaultDepth    16
    SubSection "Display"
        Depth     16
        Modes     "1024x768" "800x600" "640x480"
    EndSubSection
EndSection

Section "DRI"
    Group       0
    Mode        0666
EndSection
```

1.5 X Window 程序设计简介

1.5.1 Xlib 编程

Xlib 是对底层 X 协议的封装，作为 X 窗口系统低级库函数，它提供了许多操作 X 窗口屏幕的函数。基于 Xlib 编程是很繁琐的，Xlib 之上则是 Xt 内部库（Intrinsic library），它是面向对象的设计实

现，提供了大量的 widget。在 Xt 之上则是各种编程工具，如 Motif 和 Qt 等。

X Window 程序设计基于事件驱动机制，用户和应用程序之间有较好的交互。当应用程序初始化后，就进入一个无限的事件循环。如果有事件，则调用相关函数来处理。

在 X11/Xlib.h 中使用 Xlib 库编程用到的特定结构定义，说明如表 1-2 所示。

表 1-2 使用 Xlib库编程用到的特定结构

结构名	功能描述
Display	其中包含有关 X 服务器与屏幕的信息，当应用程序与 X 服务器相连时，才为该结构赋值
Screen	显示屏幕信息
Window	用来标识 X 窗口，由 XCreateWindos()或 XCreateSimpleWindow()函数产生并返回一个应用程序一个窗口
Xevent 联合结构	表示一个事件类型，事件分为暴露事件、改变尺寸事件或鼠标事件和用户操作事件
Gc	代表图形上下文标识符，其中包含一些画图的信息，它控制了所画图形的前景色、背景色、线条宽度和线条形式等要素
XGCValue	表示图形上下文设置变量

运行 X 程序时需要指定 Display 方法如下：

```
hostname : display_number.screen_number
```

示例：Xlib 应用程序

下面是一个简单的 Xlib 应用程序，其功能是在窗口中打印“hello world”。该程序虽然简单，但却清晰地说明了应用程序的建立过程。应用程序首先连接 X 服务器，初始化显示器。然后创建并设置窗口属性，映射窗口，进入事件循环。

```
#include <X11/Xlib.h>
#define HELLO "hello world"

main(int argc, char **argv)
{
     Display *display;        // X Server connection
     Window *win;             //  Window ID

     GC  gc;                  // Graphics context
     int bw = 2               //窗口边框的宽度
     int bc = BlackPixel;     //窗口边框的颜色值
     int bgc = WhitePixel;    //窗口的背景颜色值
     XSizeHints  hints;       // Window sizing hints
     XEvent  event;           //事件数据结构

     /*打开显示，即连接 X服务器，初始化显示器。参数为 NULL,表示自动采用用户的 UNIX 环境变量 DISPLAY 来定义*/
     display = XOpenDisplay(NULL);

     //设置窗口尺寸和位置的 hints
     hints.x = 200; hints.y = 200;
     hints.width=300; hints.height=200;
     hints.flags= PPosition| PSize;

     //创建窗口
     win = XCreateSimpleWindow(display, DefaultRootWindow(display), hints.x, hints.y,  hints.width,
hints.height, bw, bc, bgc);

     XSetStandardProperties(display, win, HELLO, HELLO, argv, argc, &hints);

     //为写文本创建图形上下文
     gc = XCreateGC(dispaly, win, NULL, NULL);
```

```
    //定义感兴趣的事件（仅暴露事件）
    XSelectInput(display, win, ExposureMask);

    //映射为可以显示出来的窗口，即显示窗口
    XMapWindow(display, win);

    //事件循环
    while (TRUE) {
       XNextEvent(display, &event) ;
       //除了最近的暴露事件外，删除其他所有事件
       if (event.type == Expose && event.xexpose.count == 0){
            int x, y;
            while (XCheckTypedEvent(display, Expose, &Event)); //  检查事件类型
            XClearWindow(display, win);    // 清除窗口
            //画出字符串“hello world”
            XDrawImageString(display, win, gc, 50, 50, HELLO, strlen(HELLO));
     } //End if
   } // end while

   exit(0); // 退出应用程序
}
```

1.5.2 Motif 编程

Xlib 库是用 C 语言实现的底层函数库，编程复杂且不灵活。 MIT 和 Digital 公司推出基于 Xlib 的工具包，比使用 Xlib 的效率高。Motif 是开放软件基金会 OSF在 Xlib 和 Xt 上建立的更高层的工具包，称为“ Motif 库 ”或‘ Xm 库 ”。

Motif 实现了一系列方便用户操作的组件，如按钮、菜单和对话框等，可以改变系统中各种组件的外观特性。由于 Motif 涉及 Xt和其定义复杂的 Widget，因此使用 Motif编程仍然很困难。

示例：Motif应用程序

下面是一个简单的 Motif 应用程序，其功能是按下按钮时打印字符串 “Hello World!” 。该程序首先建立顶层窗口，然后创建按钮，设置回调函数，显示窗口并进入事件循环。从这个例子可以看出，Motif 与 Xlib 编写应用程序的过程类似，但简单一些。

```
/*File: helloworld.c*/
#include <Xm/Xm.h>
#include <Xm/PushB.h>

void callback(Widget w, Xtpointer client_data, XmPushButtonCallbackStruct *cbs)
{
printf( "Hello world!\n ");
}

main(int argc, char **argv)

{
int n;
Arg args[10];
Widget topplevel, button;
XtAppContext  app;
XmString label;

   //创建项层窗口
  topplevel = XtVaAppInitialize(&app, "Push", NULL, 0,
      &argc, argv, NULL, NULL);

   //建立按钮标签的字符串
```

```
    label = XmStringCreatSimple("Hello Word! ");

    //建立按钮
    n = 0;
    XtSetArg(args[n], XmNlabelSting, label);
    n++;
    button = XmCreatePushButton(topplevel, "Push_me", args, n);
    XtManageChild(button);
    XmStringFree(label);

      //为 widget 加上回调函数
    XtAddCallback(button, XmNactivateCallback, (XtCallbackproc)callback, NULL);

    XtRealizeWidget(topplevet);        //显示窗口
    XtAppMainLoop(app);              //进入事件循环处理
}
```

1.5.3 GTK/GNOME

GNOME（GNU Network Object Model Environment）是一个基于 UNIX 的桌面系统，GTK 是 GIMP Toolkit 的缩写，它是编写图形界面比较流行的高层函数库之一。 GIMP（GNU Image Manipulation Program）是 Linux/Unix 中的高级图形处理软件，GTK 使用 C 语言编写，即只提供 C 的函数调用接口。

GTK 基本上不使用 Xlib 库函数，而是使用更底层的函数库 GDK 和与界面无关的函数库 GLIB，从而使其可以方便地移植到其他操作系统上。

示例：GTK+程序编程

下面是一个简单的 GTK/GNOME 应用程序，其功能是在单击按钮时显示字符串 “Hello World”：

```
#include <gtk/gtk.h>

//这是一个回调函数，在这个例子中参数被忽略
void hello( GtkWidget *widget,  gpointer data )
{
   g_print ("Hello World\n");
}

gint delete_event( GtkWidget *widget,  GdkEvent  *event,  gpointer  data )
{
   /* 如果在“delete_event ”事件处理函数中返回 FALSE，GTK将发出“ destroy ”信号。返回 TRUE，表示不想销毁这个窗口，这对于想弹出 “are you sure you want to quit?” '    类型对话框来说是有用的 */
    g_print ("delete event occurred\n") ;

    return(TRUE);
}

/*另一个回调函数，销毁窗口，退出应用程序 */
void destroy( GtkWidget *widget,
           gpointer   data )
{
   gtk_main_quit();
}

int main( int   argc,
        char *argv[] )
{
   /* GtkWidget 是widget 的类型 */
   GtkWidget *window;
   GtkWidget *button;

   //分配命令行参数，初始化 GTK
```

```
    gtk_init(&argc, &argv);

    //创建一个新的窗口
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /*窗口管理器关闭窗口时，发出 "delete_event" 信号，将调用上面定义的 delete_event() 函数*/
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);

    /* 这里我们连接' destroy" 事件到信号处理函数。在窗口上调用 gtk_widget_destroy() 函数或者在' delete_event '回调函数
里返回 FALSE时，这个事件发生*/
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);

    /* 设置窗口边框的宽度。*/
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /*创建新的带有标签 "Hello World" 的按钮。 */
    button = gtk_button_new_with_label ("Hello World");

    /* 当按键接收到" clicked "信号时，它将调用上面定义的函数 hello () * /
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (hello), NULL);

    /*当" clicked" 时，通过调用 tk_widget_destroy(window) 函数引起窗口被销毁。detroy错误号来自这里或窗口管理器*/
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC (gtk_widget_destroy),
                               GTK_OBJECT (window));

    /*把按钮放到窗口中*/
    gtk_container_add (GTK_CONTAINER (window), button);

    /* 最后的步骤是显示这个新创建的 widget*/
    gtk_widget_show (button);

    /* 显示窗口*/
    gtk_widget_show (window);

    /*所有的 GTK applications 必须有一个 gtk_main(), 它进入事件循环，处理发生的事件*/
    gtk_main ();

    return(0);
}
```

CORBA（Common Object Request Broker Architecture）是一组标准，用来定义“分布式对象系统”，其目的是定义一套协议。符合这个协议的对象可以互相交互，与具体的编程语言和操作系统无关。

CORBA 制定了一套对象间通信的协议，通信介质被称为“ORB”（Object Request Broker），用于在对象之间传递消息。如果对象在同一台机器上，ORB 可以采用一些 IPC 技术来优化消息的传递；如果在不同的机器上，则使用 IIOP（Internet Inter-ORB Protocol）或 GIOP 协议（可以建立在任何网络通信协议之上）。IIOP 是基于 IP 协议，为方便 Internet 上的 CORBA 应用而设计的。

ORB 是 CORBA应用的基础，不同的公司可以开发出不同的 ORB 但由于使用相同的通信协议，因此基于不同 ORB 实现的应用之间可以无缝通信。

为了允许不同语言（机器环境和操作系统）下的对象之间可以对话，需要一套共同语言，IDL（Interface Definition Language）专门为此而制定。IDL 看起来非常类似 C++（或 Java）的头文件定义，与 DCOM 下的 IDL 并不是一回事。但可以想像，它们非常接近。

IDL 定义了一个对象支持的方法和属性，一个对象的实现只要符合这个定义，即可完全自由开发。例如可以用任何语言给出一个对象的 IDL 文件，其他对象就完全掌握了这个对象的行为，可以按照自己的意图随意操纵。

GNOME 中 CORBA ORB 的实现称为“ORBit”，它是按照 CORBA 2.2 标准写的。特别是其 hooks 使得与 GNOME 应用可以很好地交互。ORBit 用 C 编写，非常小而且运行非常快。

Bonobo 是一个组件系统，其中提供了定义完善的接口，用于与其他组件协同工作。在 Bonobo 体系中，CORBA 是将组件联系在一起的通信层。使用 CORBA 作为底层的另一个好处是组件及使用组件的软件可以用任何语言编写而不影响它们的交互。

Bonobo 的目的是为 Unix 平台提供一套组件编写和符合文档的标准环境， GNOME 只是其中的一个实现。

1.5.4 KDE

KDE 是 Linux 上第 1 个真正的桌面，由 KDE 桌面环境（包括 WM）、KDE 应用开发平台和 KDE 办公套件 3 大部分组成，其体系结构如图 1-5 所示。

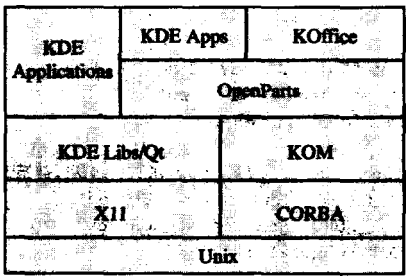


图 1-5 KDE 的体系结构

Qt 是跨平台且基于 C++语言的高层图形库，完全符合面向对象设计。 KDE 是基于 Qt 编写的窗口管理器，它包含了面板、桌面管理和文件管理等一系列的应用软件。作为最受欢迎的桌面环境之一，KDE 提供给开发者的开发平台类似于微软的 MFC/COM/ActiveX（对应于 QT/KOM/OpenParts），允许 RAD 和对现有组件及技术的充分利用。

1.6 显示驱动程序

1.6.1 显卡驱动方式

显卡驱动方式有两种，一种是使用厂商提供的驱动程序模块，这是最好的方式；另一种是使用通用帧缓冲（FrameBuffer）驱动方式。

在运行厂商提供的驱动程序时，CPU 几乎不参与显示任务，而专注于运算工作。例如，如果显卡是 nvidia 的，XFree 默认的驱动是 nv。它是一个 nvidia 显卡的通用驱动，不能充分发挥 nvidia 显卡的性能。这个 nv 驱动模块可以在/usr/X11R6/lib/modules/drivers 目录下找到，文件名是 nv_drv.o。

XF86Conf 文件中的 Section module 小节指定使用的驱动程序，该小节代码的示例如下：

```
Section "Module"
      Load  "dbe"
      Load  "extmod"
      Load  "fbdevhw" #这里用的是 fbdev
      Load  "glx"
      Load  "record"
      Load  "freetype"
      Load  "type1"
      Load  "dri"
EndSection
```

在 XFree86 4.0 以上版本中，X 主程序和驱动模块是分开的，主程序是/usr/X11R6/bin/XFree86（即上层部分），主程序启动的时候根据配置文件决定使用哪一个驱动模块。

（1）安装显卡驱动程序

许多 3D 显卡有 Linux 版本驱动程序，可从其厂商的网页上得到。例如，一个驱动程序为 XF86_SVGA.tar.gz。在 Linux 下解压缩得到 XF86_SVGA 文件，将该文件拷贝到/usr/X11R6/bin 目录下。

运行 Xconfigurator 配置程序，在显卡列表中选择加载的显示模块 SVGA。然后选择显示器型号、显存大小及刷新频率，重新输入 startx 启动 X 窗口。

（2）通用帧缓冲驱动方式

Linux 2.2.x 以上版本提供了 frame buffer 设备，通过 VESA VBE 2.0 标准，利用显卡的 SVGA 特性，配合 XFree86 的 XF86 FBDev 可驱动任何显卡。

1.6.2 帧缓冲

帧缓冲是一个提供显示内存和显示芯片寄存器从物理内存映像到进程地址空间中的设备，是用户进入图形界面很好的接口。在使用帧缓冲时， Linux 将显卡置于图形模式下，系统启动时会有一个小企鹅 logo。

通过帧缓冲设备可以直接对显存进行读写操作，用户不必关心物理显存的位置及换页机制等具体细节。对于用户而言，它和/dev 下面的其他设备没有什么区别。

帧缓冲模式的显卡不参与任何运算，它只是把 CPU 将运算后的结果存放在显存中。然后送到屏幕显示，因而 CPU 负担很重。

假定 framebuffer 的设备文件是/dev/fb，应用程序可以通过下面的命令清空屏幕，屏幕将全变黑。通过移动鼠标或切换窗口等操作可恢复原来的显示：

```
#dd if=/dev/zero of=/dev/fb
```

应用程序可以通过下面的命令将帧缓冲中的内容读出到文件 fbfile 中：

```
#dd if=/dev/fb of=fbfile
```

应用程序可以通过下面的命令将文件 fbfile 中的内容写入到帧缓冲中：

```
#dd if=fbfile of=/dev/fb
```

帧缓冲驱动程序对应的源文件在 linux/drivers/video/ 目录中，总的抽象设备文件为 fbcon.c，在这个目录下还有与各种显卡驱动相关的源文件。

帧缓冲驱动程序提拱了 ioctl 命令，通过这些命令可以获得显示设备的一些固定信息（如显示内存大小及显示芯片信息）、与显示模式相关的可变信息（如分辨率、像素结构及每扫描线的字节宽度），以及伪彩色模式下的调色板信息等。应用程序可以将 PCI 设备的内存 I/O（memio）映像到进程地地址空间，这样即可直接访问这些寄存器。这些 memio 一般是用来控制显示卡的寄存器，通过对这些寄存器的操作，应用程序即可控制特定显卡的加速功能。

1.6.3 在台式机上使用通用帧缓冲

台式机上的 Redhat 9.0 操作系统默认时没有添加对 framebuffer 设备的支持，因此需要重编译内核，添加 framebuffer 设备模块，具体操作步骤如下。

（1）配置内核

运行命令 make menuconfig，选择“Code maturity level option”并设置“Prompt for development and/or

incomplete code/drivers”。选择“Console drivers”，设置“ Support for framebuffer device”为 built-in（作为内核编译部分）。然后配置驱动程序，大多数显卡能使用“VESA VGA graphics console”，选择它或与显卡匹配的驱动程序。最后激活“Advanced low level driver options”，并确认 16和 32bpp（每像素点 32位）支持被激活。

（2）编译内核

使用下面的命令：

```
make dep
make bzImage
```

新生成的内核应该在 arch/i386/boot/bzImage 中。

（3）拷贝内核到启动目录下。

使用下面的命令：

```
cp arch/i386/boot/bzImage /boot/linux.vesafb
```

（4）编译/etc/lilo.conf

注意保存一个 lilo.conf 的备份，并制作一张急救盘。这样如果出现错误，可用急救盘启动。

文件 lilo.conf 定义了系统如何启动，系统不同，文件 lilo.conf 的内容不同，下面是一个示例：

```
# LILO configuration file
boot = /dev/hda3
delay = 30
image = /boot/vmlinuz
root = /dev/hda3
label = Linux
read-only # Non-UMSDOS filesystems should be mounted read-only for checking
other=/dev/hda1
label=nt
table=/dev/had
```

制作一个新的“ image ”小节，并做如下改动：

```
# LILO配置文件
boot = /dev/hda3
delay = 30    #延迟 30秒
vga = 791    #意味着显示分辨率为 1024x768, 16bpp
image = /boot/linux.vesafb    #启动的内核 image文件
root = /dev/hda3    # 启动的根硬盘
label = Linux-vesafb  # 开机时显示的选择项
read-only  #非 UMSDOS文件系统为了检查应该被以仅读方式挂接。
image = /boot/vmlinuz
root = /dev/hda3
label = Linux
read-only  #非UMSDOS 文件系统为了检查应该被以仅读方式挂接。
other=/dev/hda1
label=nt
table=/dev/hda
```

（5）运行 lilo 程序使新的变化生效。

运行命令如下：

```
lilo
```

（6）重新启动系统，计算机在启动时应该能看到一个企鹅图标出现。

（7）下面的短程序打开 framebuffer 并绘制一个红色梯形分配的方块：

```
int main()
```

```
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;
    long int screensize = 0;
    char *fbp = 0;
    int x = 0, y = 0;
    long int location = 0;
    //打开设备文件
    fbfd = open("/dev/fb0", O_RDWR);
    if (!fbfd) {
        printf("Error: cannot open framebuffer device.\n");
    exit(1);
    }
    printf("The framebuffer device was opened successfully.\n");

    //得到固定的屏幕信息
    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
    printf("Error reading fixed information.\n");
    exit(2);
}

//得到可变的屏幕信息
if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
    printf("Error reading variable information.\n");
    exit(3);
}
printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel );

// 计算屏幕的大小 (byte)
screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
// 映射设备到内存
fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
                                        fbfd, 0);
if ((int)fbp == -1) {
    printf("Error: failed to map framebuffer device to memory.\n");
    exit(4);
}
printf("The framebuffer device was mapped to memory successfully.\n");

x = 100; y = 100; //我们开始放像素点的位置
// 计算出所放像素点在内存中的位置
for ( y = 100; y < 300; y++ )
    for ( x = 100; x < 300; x++ ) {
        location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
                    (y+vinfo.yoffset) * finfo.line_length;
        if ( vinfo.bits_per_pixel == 32 ) {
            *(fbp + location) = 100; // Some blue
            *(fbp + location + 1) = 15+(x-100)/2; // A little green
            *(fbp + location + 2) = 200-(y-100)/5; // A lot of red
            *(fbp + location + 3) = 0; // No transparency
        } else { //assume 16bpp
            int b = 10;
            int g = (x-100)/6; // A little green
            int r = 31-(y-100)/16; // A lot of red
            unsigned short int t = r<<11 | g << 5 | b;
            *((unsigned short int*)(fbp + location)) = t;
        }
    }

   munmap(fbp, screensize);
   close(fbfd);
   return 0;
}
```

1.6.4 帧缓冲设备驱动程序

帧缓冲设备提供了对显示卡硬件的抽象，代表了一些视频硬件的帧缓冲。它允许应用程序通过定义好的接口访问显示卡硬件，这样软件不必知道任何底层的寄存器等。

帧缓冲设备通过位于/dev 目录下的特殊设备节点/dev/fb*被访问。从用户角度来看，帧缓冲设备与在/dev 目录下的其他的设备一样，是主设备号为 29 的字符设备。次设备号表示帧缓冲的序号，表示如下：

```
0 = /dev/fb0    //第 1个帧缓冲
1 = /dev/fb1    //第 2个帧缓冲
...
31 = /dev/fb31 //第 32个帧缓冲
```

为了与后面的版本相适应，可创建下面的符号链接：

```
/dev/fb0current -> fb0
/dev/fb1current -> fb1
```

帧缓冲设备也是正常的内存设备，这意味着可读写其内容。例如可用下面的命令制作一个屏幕的快照：

```
cp /dev/fb0 myfile
```

也可以一次使用多个帧缓冲设备，如果有一个显卡附加内在的硬件，相应的帧缓冲设备分别是/dev/fb0 和l/dev/fb1。它们分别独立工作，应用程序（如 X Server）默认使用/dev/fb0，在旧版本中将使用/dev/fb0current 。可通过定义环境变量$FRAMEBUFFER 来定义一个替换的帧缓冲设备，在 sh/bash 中定义如下：

```
export FRAMEBUFFER=/dev/fb1
```

在 csh 中定义如下：

```
setenv FRAMEBUFFER /dev/fb1
```

X Server（XF68_FBDev）是最常见的使用帧缓冲设备的应用程序， XF68_FBDev是位映射、非加速的视频硬件用的 X Server 是 XFree86 的一部分，有两种模式。

如果配置文件/etc/X11/XF86Config 的 fbdev 驱动程序的 Display 子段设置模式为“default”, X Server 将使用默认模式，应用程序仅需要进行屏幕设置（如颜色位数和分辨率等）。分辨率由 /dev/fb0 或 $FRAMEBUFFER 决定。还必须用 Depth 关键字设置颜色深度，并使用 Virtual 关键字设置可视的分辨率。这些值在默认情况下由 XFree86 配置文件提供。

下面通过 ATI 系列显卡的帧缓冲设备驱动程序中的探测函数说明帧缓冲设备的建立过程。

在 ATI 显卡驱动程序中，函数 radeonfb_pci_register 分析得到显卡的各种信息，探测显示器的连接方式并建立 I/O 映射，然后注册帧缓冲设备。函数 radeonfb_pci_register（在 drivers/video/aty /radeon_base.c 中）分析如下：

```
static int radeonfb_pci_register (struct pci_dev *pdev,
                    const struct pci_device_id *ent)
{
    struct fb_info *info;
    struct radeonfb_info *rinfo;
    int ret;
```

```
//在 PCI 的配置中激活设备*/
ret = pci_enable_device(pdev);

//分配结构 fb info 对象空间，赋上 pdev->dev
info = framebuffer_alloc(sizeof(struct radeonfb_info), &pdev->dev);


rinfo = info->par;
rinfo->info = info;
rinfo->pdev = pdev;

spin_lock_init(&rinfo->reg_lock);
init_timer(&rinfo->lvds_timer);
  //延迟执行 LVDS 的启动/关闭
rinfo->lvds_timer.function = radeon_lvds_timer_func;
rinfo->lvds_timer.data = (unsigned long)rinfo;

  //初始化 rinfo，得到显卡相关信息
strcpy(rinfo->name, "ATI Radeon XX ");
rinfo->name[11] = ent->device >> 8;
rinfo->name[12] = ent->device & 0xFF;
rinfo->family = ent->driver_data & CHIP_FAMILY_MASK;
rinfo->chipset = pdev->device;
rinfo->has_CRTC2 = (ent->driver_data & CHIP_HAS_CRTC2) != 0;
rinfo->is_mobility = (ent->driver_data & CHIP_IS_MOBILITY) != 0;
rinfo->is_IGP = (ent->driver_data & CHIP_IS_IGP) != 0;

/* 设置物理基地址 */
rinfo->fb_base_phys = pci_resource_start (pdev, 0);
rinfo->mmio_base_phys = pci_resource_start (pdev, 2);

/* 请求内存区域 */
ret = pci_request_regions(pdev, "radeonfb");

/* 映射区域 */
rinfo->mmio_base = ioremap(rinfo->mmio_base_phys, RADEON_REGSIZE);

rinfo->fb_local_base = INREG(MC_FB_LOCATION) << 16;

/* 得到显存大小及位宽信息 */
radeon_identify_vram(rinfo);

rinfo->mapped_vram = min_t(unsigned long, MAX_MAPPED_VRAM
                                          rinfo->video_ram);

do {
        //将物理地址及大小映像到 CPU 的虚拟空间，以便 readw等函数能访问
        rinfo->fb_base = ioremap (rinfo->fb_base_phys
                        rinfo->mapped_vram);
} while (   rinfo->fb_base == 0 &&
       ((rinfo->mapped_vram /=2) >= MIN_MAPPED_VRAM) ) ;

if (rinfo->fb_base)  //清空
     memset_io(rinfo->fb_base, 0, rinfo->mapped_vram);

if (!rinfo->is_mobility)
     //映射 PCI ROM 到内核空间，并得到相关信息
     radeon_map_ROM(rinfo, pdev);

/* 得到显卡的 PLL相关信息 */
radeon_get_pllinfo(rinfo);

/*设置 rinfo->info 相关信息*/
radeon_set_fbinfo (rinfo);
```

```
    /* 探测显示器的连接方式及类型等信息 */
    radeon_probe_screens(rinfo, monitor_layout, ignore_edid);

    /* 创建显示模式链表，并检测出当前的模式 */
    radeon_check_modes(rinfo, mode_option);

    /* 注册设备到 sysfs 文件系统*/
    if (rinfo->mon1_EDID)
         sysfs_create_bin_file(&rinfo->pdev->dev.kobj, &edid1_attr);
    if (rinfo->mon2_EDID)
         sysfs_create_bin_file(&rinfo->pdev->dev.kobj, &edid2_attr);

    /*在切换显示模式之前操作当前模式设置 */
    radeon_save_state (rinfo, &rinfo->init_state);
    memcpy(&rinfo->state, &rinfo->init_state, sizeof(struct radeon_regs));

    pci_set_drvdata(pdev, info);

    /*注册帧缓冲设备层*/
    ret = register_framebuffer(info);
    ……
}
```

函数 register_framebuffer 注册一个帧缓冲设备，参数 fb_info 是帧缓冲信息结构。如果注册失败，则返回负值的 errno 错误信息；否则返回 0。

帧缓冲设备作为字符设备注册，其主设备号为 29。函数 register_framebuffer 在/dev 目录下生成 fb0 或 fb1 的设备文件，在 sysfs 文件系统中注册相应的设备类，并将帧缓冲信息结构 fb_info 保存到全局的结构数组中。最后调用 notifier_call_chain 函数来执行 fb_notifier_list 链表中的每个回调函数来处理帧缓冲的事件。

函数 register_framebuffer （在 drivers/video/fbmem.c 中）分析如下：

```
int register_framebuffer(struct fb_info *fb_info)
{
    int i;
    struct class_device *c;
    struct fb_event event;

    if (num_registered_fb == FB_MAX) // FB_MAX 为最大 32 个帧缓冲设备
         return -ENXIO;
    num_registered_fb++;
    for (i = 0 ; i < FB_MAX; i++)
         if (!registered_fb[i]) // fb_info 结构数组元素为 0
              break;
    fb_info->node = i;

    //加字符设备类型 fb class 及设备结构到 sysfs 文件系统中
    c = class_simple_device_add(fb_class, MKDEV(FB_MAJOR, i),
                   fb_info->device, "fb%d", i);
    …

    if (fb_info->pixmap.addr == NULL) {
         fb_info->pixmap.addr = kmalloc(FBPIXMAPSIZE, GFP_KERNEL);
         if (fb_info->pixmap.addr) {
              fb_info->pixmap.size = FBPIXMAPSIZE;
              fb_info->pixmap.buf_align = 1;
              fb_info->pixmap.scan_align = 1;
              fb_info->pixmap.access_align = 4;
              fb_info->pixmap.flags = FB_PIXMAP_DEFAULT;
         }
    }
```

```
    fb_info->pixmap.offset = 0;

    if (!fb_info->modelist.prev ||
        !fb_info->modelist.next ||
        list_empty(&fb_info->modelist)) {
            struct fb_videomode mode;

            INIT_LIST_HEAD(&fb_info->modelist);
            fb_var_to_videomode(&mode, &fb_info->var); //将 var 转换成 mode 对应结构
            //将 mode 加入到链表 modelist 中
            fb_add_videomode(&mode, &fb_info->modelist);
    }

    registered_fb[i] = fb_info; //加入到全局的数组中
    event.info = fb_info;
    //调用在 fb_notifier_list 链表中的每个函数
    notifier_call_chain(&fb_notifier_list,
                FB_EVENT_FB_REGISTERED, &event);
    return 0;
}
```

1.7 X Window 协议

X Window 系统对数据传输、窗口管理及输入法等通过协议进行规范，并提供了 API 接口函数规范。规范分为核心和扩展协议、其他协议和接口两部分，分别说明如下。

（1）核心和扩展协议

核心和扩展协议列出如下：

- 核心 X 协议
- 核心 X协议库 API
- MIT-SHM（共享内存）扩展
- XC-MISC 扩展
- BIGREQ（大请求)扩展
- 形状（非矩形窗口）扩展协议
- 形状（非矩形窗口）扩展库 API
- XC-APPGROUP（应用程序组)扩展
- 安全扩展
- TOG CUP （颜色映射实用策略）扩展
- DPMS （显示电源管理信令）扩展协议
- DPMS （显示电源管理信令）扩展客户端库 API
- EVI（扩展的可视信息)扩展
- Xi（X 输入）扩展协议
- Xi（X 输入）扩展库 API
- Xi（X 输入）Porting Information [PDF]
- 双缓冲扩展协议
- 双缓冲扩展库 API
- 记录扩展协议
- 记录扩展库API
- XTEST 扩展协议
- XTEST 扩展库 API

- 同步扩展协议
- 同步扩展库 API
- XPrint 扩展协议
- XPrint 扩展库 API
- XKEYBOARD扩展协议
- XKEYBOARD 扩展库 API

（2）其他协议和接口

其他协议和接口列出如下：

- X传输协议
- XDMCP（X Display Manager Control Protocol，X 显示管理器控制协议）
- 字体服务器协议
- XIM（X 输入法）协议
- XIM（X 输入法）传输
- ICE（客户端交换)协议
- ICE（客户端交换)库 API
- ICCCM（Inter-Client Communcations Conventions Manual，客户端之间通信规范手册）
- SM（X会话管理）协议
- SM（X会话管理）库 API

1.7.1 X 协议

Xlib 是一个 C 语言子例程库，应用程序（客户端）用其通过流连接来与窗口系统进行交互。一个 X Server 能为多个显示屏提供服务，单个用户的一套显示屏以及一个键盘和一个指针设备（通常为鼠标）组成在一起被称为“display”。X 为窗口提供了图形、文本和光栅操作，还为图形对象提供了离开屏幕的存储，这个存储被称为“位图”。Xlib 的大多数函数仅增加请求到一个输出缓冲中，这些请求后来在 X Server 上被异步执行。许多 Xlib 函数将返回一个整数资源 ID，这个 ID 允许引用存储在 X Server 上的对象，这些对象可能是定义在<X11/x.h>中的 Window、Font、Pixmap、Colormap、Cursor 和 GContext类型。

在程序能使用 display 前必须建立到 X Server 的连接，一旦建立了一个连接，那么能使用 Xlib 宏和函数来得到 display 的信息。display 指包括输入设备和显示屏设备的软件模型，相当于 X Server的终端功能部分。

X协议是会话层的协议，它解释了 X客户端与 X服务器通信的数据格式，并定义了各种命令关键字。X协议被封装成了 Xlib 函数库，X 协议的解析工作由 Xlib 函数库完成。这样，X 客户端应用程序直接调用 Xlib 库函数即可完成图形显示等工作，而不必关心 X协议的细节。

由于 X 协议条目很多，因此这里只说明了 X协议的格式及连接初始化时信息内容。具体内容请参考 X协议规范《X Window System Protocol X Consortium Standard X Version 11, Release 6.7 DRAFT》。

1. 协议格式

（1）请求格式

每个请求装有一个 8位主操作码和一个 16位长度的域，这个域以 4个字节为单位。每个请求由 4个字节的头（装有主操作码、域长度和一个数据字节），后面跟 0 或更多的附加数据字节。长度域定义了整个请求的长度，包括头。在一个请求中的长度域必须等于容纳请求要求的最小长度。主操作码 128~255 保留作为扩展使用，扩展可被用来容纳多个请求。扩展请求通常有一个附加的次操作码被编码放在请求头的第 2个字节，然而这个次操作码的位置和扩展请求的其他域的解释在 X 协议中没有定

义。每个给定连接上的请求都指定了一个系列号，从 1 开始，这些请求用在 Reply（回答）、Error（错误）和 Event（事件）中。

（2）Reply 格式

每个 Reply 装有一个 32 位长度的域，以 4 个字节为单位进行表达。每个 Reply 由 32 个字节和跟随的 0 或附加数据字节组成，在长度域中定义。不用的字节并不保证是 0，每个 Reply 至少装有相应请求签名的 16 位序列数。

（3）Error 格式

错误报告 Error 为 32 个字节，每个包括 8 位错误代码。错误代码从 128~255 保留作为扩展使用。每个 Error 还包括失败请求的主和次操作码，以及至少装有相应请求签名的 16 位序列数。对于 Colormap、Cursor、Drawable、Font、GContext、IDChoice、Pixmap 和 Window，这些错误还返回失败的资源 ID。对于 Value 错误，错误值被返回，其他的核心错误不返回附加数据。在 Error 中没有用的数据并不保证为 0。

（4）Event 格式

Event 为 32 个字节，每个装有 8 位类型码。如果事件从 SendEvent 请求中产生，最重要的事件将被设置。事件码从 64~126 保留作为扩展使用，核心协议没有定义这些事件的机制。每个核心事件（除了 KeymapNotify）还至少装有上一个被客户端发出的请求签名的 16 位序列数，这个客户端正被服务器处理。

2. 连接建立

对于远程客户端来说，X 协议可在任何可靠的字节流上建立。

（1）连接初始化

客户端必须发送一个初始的数据字节鉴别被使用的字节序，这个字节值必须是八进制数的 102 或 154。值 102（ASCII 的大写 B）意味着首先被传送最重要的值，值 154（ASCII 小写的 l）意味着首先传送最不重要的字节。除了明确在协议中标明的，所有被客户端发送的 16 位和 32 位数据都按这个字节值传送，并且所有被服务器返回的 16 位和 32 位数据也都按此字节序传送。

遵循这个字节序，客户端在连接建立时发送下面信息。

- 协议主版本号：16 位无符号整型。
- 协议次版本号：16 位无符号整型。
- 授权协议名：STRING8（8 位字符串）。
- 授权协议数据：STRING8。

版本号表示客户端期望服务器使用的协议的版本号。

authorization 名字表示客户端期望服务器使用哪种授权协议和协议定义的数据，合理的授权机制的定义不是 X 协议的部分。一个未使用客户端期望的授权协议或仅使用主机的授权机制的服务器可以简单忽略这个信息。如果 name 和 data 字符串为空，则将被解释为“没有明确的授权协议”。

（2）服务器响应

在连接建立时客户端接收到下面的返回的 success 值，其取值为：

```
success: {Failed, Success, Authenticate}
```

如果返回的 success 值是 Failed，并且连接没有被建立，客户端收到下面的附加数据。

- 协议主版本号：16 位无符号整型。
- 协议次版本号：16 位无符号整型。
- 原因：STRING8（8 位字符串）。

如果返回的 success 值是 Authenticate（授权），客户端收到下面的附加数据，并且授权谈判是需要的。

原因：STRING8（8位字符串）。

原因字符串的内容由使用的授权协议来定义，授权谈判的语义未被限制，除了谈判必须最终用一个来自服务器的装有 Failed或 Success的 success值的回答来结束。

如果返回的 success值是 Success，客户端收到下面的信息（其中包括服务器、显示屏和可视类型信息),并且连接被成功建立：

```
protocol-major-version: CARD16
protocol-minor-version: CARD16
vendor: STRING8
release-number: CARD32
resource-id-base, resource-id-mask: CARD32
image-byte-order: {LSBFirst, MSBFirst}
bitmap-scanline-unit: {8, 16, 32}
bitmap-scanline-pad: {8, 16, 32}
bitmap-bit-order: {LeastSignificant, MostSignificant}
pixmap-formats: LISTofFORMAT
roots: LISTofSCREEN
motion-buffer-size: CARD32
maximum-request-length: CARD16
min-keycode, max-keycode: KEYCODE
where:
     FORMAT: [depth: CARD8,
              bits-per-pixel: {1, 4, 8, 16, 24, 32}
              scanline-pad: {8, 16, 32}]
     SCREEN: [root: WINDOW
              width-in-pixels, height-in-pixels: CARD16
              width-in-millimeters, height-in-millimeters: CARD16
              allowed-depths: LISTofDEPTH
              root-depth: CARD8
              root-visual: VISUALID
              default-colormap: COLORMAP
              white-pixel, black-pixel: CARD32
              min-installed-maps, max-installed-maps: CARD16
              backing-stores: {Never, WhenMapped, Always}
              save-unders: BOOL
              current-input-masks: SETofEVENT]
     DEPTH: [depth: CARD8
              visuals: LISTofVISUALTYPE]
     VISUALTYPE: [visual-id: VISUALID
              class: {StaticGray, StaticColor, TrueColor, GrayScale
              PseudoColor, DirectColor}
              red-mask, green-mask, blue-mask: CARD32
              bits-per-rgb-value: CARD8
              colormap-entries: CARD16]
```

3. Xlib的函数功能分类说明

（1）Display 函数

在应用程序建立到 X Server 的连接后，才能使用 Xlib 中的函数。Display的函数功能包括打开（连接到）Display 以获得 display 的信息、图像格式或 screen；产生 NoOperation 协议请求；释放客户端产生的数据；关闭（或断开连接）一个 display;使 用 X Server 的连接关闭操作，通过线程使用 Xlib 加锁或解锁 Display，以及使用内部连接等。

使用内部连接的情况是除了连接到 X Server外，一个 Xlib 应用可能要求连接到其他种类的服务器（如输入法服务器)。使用多个 display 与其他的输入一起使用 display 的工具和客户端需要获得这些附加的连接来正确阻塞直到输入是可用的，并在输入可用时处理输入。

（2）窗口函数

在 X Window 系统中，窗口是一个在屏幕上允许用浏览图形输出的矩形区域，客户端应用程序可以在一个或多个屏幕（被一台或多台机器的 X Sever 驱动）上显示重叠的多个窗口。 Xlib 与窗口相关

的函数有创建/销毁窗口、映射/取消映像窗口、配置窗口（如窗口大小及边框等）、改变窗口重叠次序，以及改变窗口属性等。

（3）窗口信息函数

在连接到 X Server 的 display 并创建一个窗口后，能使用窗口信息函数来获得窗口的信息、转换屏幕坐标、处理属性链表、获得并改变窗口属性，以及处理窗口选中的部分属性等。

（4）颜色管理函数

每个 X Window 总有一个相关的颜色映射提供显示在屏幕上的像数和颜色值， Xlib 提供了函数处理颜色映射。X 协议使用 RGB 颜色空间的值定义颜色，RGB 颜色空间依赖于设备，在不同的设备上产生一个 RGB 值将导致不同的颜色。Xlib 为了使不同的设备有相同的颜色而提供了使用设备独立的颜色空间，它支持派生于 CIE XYZ 颜色空间的与设备无关的颜色空间，其中包括 XYZ、xyY、L*u*v* 和 L*a*b* ，以及 TekHVC颜色空间。颜色管理方面的函数功能有创建、拷贝和销毁一个颜色映像；通过名字或值指定颜色；分配、修改和释放颜色单元；读颜色映射中的条目，在颜色空间之间的转换；查询屏幕颜色的颜色级别，以及增加一个新的颜色空间等。

（5）图形上下文函数

当在 X 中执行图形操作时，很多资源被使用，正在执行的图形（如前景颜色、背景颜色和线条样式等）存储在被称为“图形上下文”（graphics contexts，GC）的资源中。大多数的图形操作把 GC 作为一个参数，虽然理论上 X 协议允许在应用程序之间共享 GC 但执行操作时，应用程序使用自己的 GC是理想的。共享 GC 是不鼓励的，因为库可能缓存 GC 状态。窗口或位图能进行图形操作时需要使用 GC 一个指定的屏幕创建一个 GC。

图形上下文函数的功能有处理图形上下文或状态，以及使用图形上下文操作函数。

（6）图形函数

一旦建立到一个 display 的连接，则可以使用 Xlib 图形函数，这些图形函数的功能有清除和拷贝区域，画点、线、矩形或弧线，填充区域，处理字体，绘制文本，以及在客户端和服务器间传输图像等。如果与 GC 有同样的绘制函数，则使用 GC 的函数。

（7）窗口和会话管理器函数

Xlib提供了窗口管理器或会话管理函数，它们的功能有改变一个窗口的父窗口、控制一个窗口的生存期、管理安装的颜色映射、设置和返回字体搜索路径、捕获服务器、杀死一个客户端、控制屏幕保护，以及控制主机访问等。

（8）事 件

一个客户应用程序与 X Server 通过使用 XOpenDisplay 建立的连接进行通信。客户应用程序在这个连接上发送请求到 X Server，这些请求被在客户端使用的 Xlib 函数创建。许多 Xlib 函数引起 X Server 产生事件，并且用户的点击或移动指针能异步地产生事件， X Server 在同一连接返回事件到客户端。与事件相关的有事件类型、事件结构、事件掩码及事件处理。事件的类型很多，如键盘事件、鼠标事件及焦点事件等。

Xlib 事件处理函数的功能有选择事件、处理输出缓存和事件队列、从事件队列中选择事件、发送和得到事件，以及处理协议错误等。一些工具使用自己的事件处理函数，不允许改变 Xlib 中这些事件的处理函数。

大多数应用程序是事件循环，即等待一个事件，由事件决定执行的操作并执行操作相关的代码。从而导致屏幕显示的改变，然后等待下一个事件。

（9）输入设备函数

能使用 Xlib输入函数设置输入设置，这些函数的功能有捕获指针设备或指针设备中的单个按钮、捕获键盘和键盘上的单个键、恢复事件处理、移去指针设备、设置输入焦点、处理键盘和指针设备的设置，以及处理键盘编码等。

（10）本地化和国际化文本函数

一个国际化的应用程序能适应不同的国家语言、本地化定制并字符串编码，这种适应处理被称为“本地化”。程序国际化的目标是允许程序代码不用修改和再编译即允许本地化。

作为本地化机制之一，Xlib 为国际化的文本输入提供了一个 X 输入法（XIM）函数接口，为国际化的文本输出提供了一个 X 输出方法（XOM）函数接口。

在 X 中的国际化基于一个 local 的概念，它定义了程序在运行时的本地化的行为。local 在这些方面影响着 Xlib，即输入法字符的编码和处理、资源文件和值的编码、文本字符串的图像和编码，以及客户端之间文本通信的编/解码等。不同的语言有不同的编码，基至同一语言中的同一字符有不同的编码。

本地化和国际化文本函数的功能有 X local 管理、local 的设置与修改，以及输入方法和输出方法等。

（11）客户端之间通信函数

ICCM（Inter-Client Communication Conventions Manual，客户端之间通信规范手册）详细说明了 X 联盟同意的客户端之间通信规范，这些规范确保了对等的客户端、窗口和会话管理器在使用选择、剪切缓存和共享资源方面的协作关系。

Xlib 提供了许多与 ICCM 相适应的标准属性和编译接口。

Xlib 的选择和剪切缓冲机制提供了主要的编程接口，通过它，同等的客户端应用程序彼此间通信。客户端之间通信函数的功能有客户端到窗口管理器的通信、客户端到会话管理器的通信，以及标准的颜色映射等。

（12）资源管理器函数

一个程序通常需要各种 X 环境选项（如字体、颜色和光标等），在命令行指针中使用所有这些选项是很难的。因为用户可能想定制程序的许多方面，为此需要一个方便的方法来建立对程序的设置，资源管理器提供了这方面的功能。

资源定义常存在于可读的文件和服务器属性中，资源管理器是被用来管理资源的一个数据库管理器，资源管理器函数功能有创建存储数据库和查询资源等。

（13）应用程序实用函数

一旦初始化了 X 系统，即可使用 Xlib 实用函数，这些函数的功能有键盘应用、Latin-1 键盘事件处理、分配永久的存储、窗口几何尺寸分析、处理区域、剪切缓冲管理、决定合适的可视类型、处理图像、处理位图，以及上下文管理器等。

1.7.2 X 传输接口

X 传输接口把传输协议原型 API 函数转接到具体的传输协议的传输 API 函数上，这个原型 API 函数被 X Window 系统的所有的库、客户和服务器使用。除了 X 传输接口代码外，这个 API 允许源代码在没有任何变化的情况下增加新的传输类型并支持新平台。这个接口提供了足够的功能来支持所有的协议类型，包括面向连接的协议（如 X11 和 FS）和面向较少连接的协议（如 XDMCP）。

X 传输接口的源代码文件说明如表 1-3所示（在 xc/lib/xtrans 目录下）。

表 1-3 X传输接口的源代码文件说明

文件名	功能说明
Xtrans.h	与具体传输协议无关 API 的函数原型定义，是与上层协议的接口 API
Xtrans.c	与具体传输协议无关 API 的函数实现，是与上层协议的接口 API
Xtransint.h	仅被接口使用，装有接口内部使用的数据结构定义，如_Xtransport 和_XtransConnInfo 结构定义

续表

文件名	功能说明
Xtranssock.c	依赖于 Internet 协议簇传输协议的 Socket API，将接口函数原型转接到具体传输协议的 socket API 上（如：IPV6 协议的 socket API）
Xtranstli.c	依赖于 TLI 协议簇传输协议的 API
Xtransdnet.c	依赖于 DECnet 协议簇传输协议的 API
Xtranslcl.c	依赖于 SYSV 本地连接的 API
Xtransutil.c	使用在 X 传输接口的实用函数

X Window 的客户机/服务器通过 X 传输接口进行通信，客户机/服务器的应用层根据各种应用协议（如 XDCMP 等）将请求或服务封装成包，作为负载在 X 传输接口中进行传输。X 传输接口经转接后使用具体协议的传输接口来传输包， X 传输接口的协议层次如图 1-6所示。

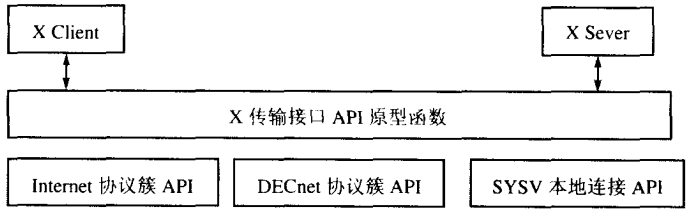


图 1-6 X传输接口的协议层次

X 传输接口是对各种具体的传输协议的封装，它提供了应用程序使用的 API。这个接口的原型 API 函数执行所有传输和系统的共有工作，如分析一个地址的语法，以获得一个主机和端口序号，这个 API 的函数调用传输特定函数。

所有协议的传输函数簇放在一个全局表 Xtransports[] 中，这个表定义了每种传输协议的入口。每个入口是传输函数簇指针，这些指针指向给定传输的接口的函数。这个 API 不提供 select()或 poll()的抽象，这些函数是传输独立的。

当应用层协议调用 X传输接口的函数原型时，X 传输接口的函数原型根据地址分析出协议名。并根据协议名在全局静态的传输表 Xtransports[]中查找到具体的传输协议函数簇，然后调用具体传输协议的函数完成数据的传输。

1．Xtransport传输表

各种协议组成一个全局静态的传输表 Xtransports[]，传输表是传输结构 Xtransport 的数组，表中每一项是代表一种传输协议的 Xtransport 结构。Xtransports[]（在 xc/lib/xtrans/Xtrans.c 中）中的一部分列出如下：

```
static Xtransport_table Xtransports[ ] = {
......
#if defined(TCPCONN)
   { &TRANS(SocketTCPFuncs), TRANS_SOCKET_TCP_INDEX },
#if defined(IPv6) && defined(AF_INET6)
   { &TRANS(SocketINET6Funcs),    TRANS_SOCKET_INET6_INDEX },
#endif /* IPv6 */
   { &TRANS(SocketINETFuncs),TRANS_SOCKET_INET_INDEX },
#endif /* TCPCONN */
......
}
```

下面以传输表 Xtransports[]的 TRANS（SocketTCPFuncs）项来说明具体的 X 传输接口，TRANS（SocketTCPFuncs）是基于 socket 的各种操作函数的集合，这些函数的实现与 socket 的应用编程类似。

若需要了解 socket 编程方法，请参看网络编程方面的书籍。 TRANS（SocketTCPFuncs）部分列出如下（在 xc/lib/xtrans/ Xtranssock.c 中）：

```
Xtransport     TRANS(SocketTCPFuncs) = {
     /* Socket Interface */
     "tcp",
         TRANS_ALIAS,
#ifdef TRANS_CLIENT
     TRANS(SocketOpenCOTSClient),
#endif /* TRANS_CLIENT */
......
#ifdef TRANS_CLIENT
     TRANS(SocketINETConnect),
#endif /* TRANS_CLIENT */
     TRANS(SocketBytesReadable),
     TRANS(SocketRead),
     TRANS(SocketWrite),
     TRANS(SocketReadv),
     TRANS(SocketWritev),
     TRANS(SocketDisconnect),
     TRANS(SocketINETClose),
     TRANS(SocketINETClose),
     };
```

2. X 传输接口与具体传输协议函数连接

下面以函数 TRANS（OpenCOTSClient）为例说明传输函数如何与具体的传输协议函数相连接。

在 Xtrans.h 中的宏定义 TRANS（func） 依据被编译的协议代码创建一个惟一的函数，这个宏的定义部分列出如下：

```
//生成函数名的宏定义，在代码被编译时设置函数名
#ifdef X11_t
#if !defined(UNIXCPP) || defined(ANSICPP)
#define TRANS(func) _X11Trans##func
#else
#define TRANS(func) _X11Trans/**/func
#endif
```

例如， TRANS（OpenCOTSClient）将变为 _X11TransOpenCOTSClient。

函数 TRANS（OpenCOTSClient）是上层协议使用的原型函数，它打开创建客户端的连接。该函数（在 xc/lib/xtrans/Xtrans.c 中）列出如下：

```
XtransConnInfo
TRANS(OpenCOTSClient) (char *address)

{
   PRMSG (2,"OpenCOTSClient(%s)\n", address, 0, 0);
   return TRANS(Open) (XTRANS_OPEN_COTS_CLIENT, address);
}
```

函数 TRANS（Open）打开一个连接，它根据地址分析得到协议。并且根据协议从全局传输表 Xtransports[]查找得到对应协议的传输函数簇，如对 TCP 协议对应的传输函数簇为 TRANS（SocketTCPFuncs），然后调用传输函数簇的对应 open 函数。

函数 TRANS（Open） （在 xc/lib/xtrans/Xtrans.c 中）分析如下：

```
static XtransConnInfo
TRANS(Open) (int type, char *address)

{
```

```
    char        *protocol = NULL, *host = NULL, *port = NULL;
    XtransConnInfo  ciptr = NULL;
    Xtransport      *thistrans;
    ......
    /*分析地址，从地址中得到协议 protocal、主机名 host及端口 port*/
    if (TRANS(ParseAddress) (address, &protocol, &host, &port) == 0)
    {
         PRMSG (1,"Open: Unable to Parse address %s\n", address, 0, 0);
         return NULL;
    }

    /*在 Xtransports[ ]表中由协议查找到传输类型，如对协议 TCP返回 TRANS(SocketTCPFuncs) *  /
    if ((thistrans = TRANS(SelectTransport) (protocol)) == NULL)
    {//没找到协议类型
          xfree (protocol);
          xfree (host);
          xfree (port);
          return NULL;
    }

    /*建立传输连接 */
    switch (type)
    {
    case XTRANS_OPEN_COTS_CLIENT:
#ifdef TRANS_CLIEN
         //对于 tcp 协议来说， ciptr为 TRANS(SocketOpenCOTSClient)
         ciptr = thistrans->OpenCOTSClient(thistrans, protocol, host, port);
#endif /* TRANS_CLIENT */
         break;
    case XTRANS_OPEN_COTS_SERVER:
#ifdef TRANS_SERVER
         ciptr = thistrans->OpenCOTSServer(thistrans, protocol, host, port);
#endif /* TRANS_SERVER */
         break;
         ......
    }
    ......
    ciptr->transptr = thistrans;
    ciptr->port = port;            /* We need this for TRANS(Reopen) */

    xfree (protocol);
    xfree (host);
    return ciptr;
}
```

函数 TRANS（SocketOpenCOTSClient）()调用函数 socket() 来创建一个 socket，创建一个适合于客户端使用的面向连接的传输。如果函数执行成功，将返回一个透明的传输连接对象；否则将返回 NULL。

函数 TRANS（SocketOpenCOTSClient） （在 xc/lib/xtrans/Xtranssock.c 中）列出如下：

```
static XtransConnInfo
TRANS(SocketOpenCOTSClient) (Xtransport *thistrans, char *protocol,
                  char *host, char *port)
{
    return TRANS(SocketOpenCOTSClientBase)(
              thistrans->TransName, protocol, host, port, -1);
}
static XtransConnInfo
TRANS(SocketOpenCOTSClientBase) (char *transname, char *protocol,
                  char *host, char *port, int previndex)
{
    XtransConnInfo  ciptr;
    int             i = previndex;
```

```
    SocketInitOnce();  //socket 初始化次

    while ((i = TRANS(SocketSelectFamily) (i, transname)) >= 0) {
         if ((ciptr = TRANS(SocketOpen) (
              i, Sockettrans2devtab[i].devcotsname)) != NULL)
             break;
        }
        ......
         return NULL;
    }

     /*存储 index 以便以后使用 */
    ciptr->index = i;
    return ciptr;
}
```

函数 TRANS（SocketOpen）调用函数 socket创建一个 socket，调用函数 setsockopt 操作 TCP。函数 TRANS（SocketOpen）列出如下：

```
static XtransConnInfo  TRANS(SocketOpen) (int i, int type)

{
    XtransConnInfo  ciptr;
    ......
    if ((ciptr->fd = socket(Sockettrans2devtab[i].family, type,
                                Sockettrans2devtab[i].protocol)) < 0 ) {

         xfree ((char *) ciptr);
         return NULL;
    }
    ......
    return ciptr;
}
```

1.7.3 XDMCP 协议

XDMCP（X Display Manager Control Protocol）是 X 显示管理器控制协议，它为来自远程主机的请求注册服务提供了统一的自动显示机制， display 由硬件和独立于任何注册服务被请求的特殊主机的进程组成。

XDMCP 被设计来为远程多个 display 提供显示管理服务的授权访问。一个被称为“显示管理器”（Display Manager）的新网络服务器将使用 XDMCP 协议与 display 通信，来商议 X 会话的启动。协议允许 display 授权管理器，也允许大多数配置信息集中在管理器，以减轻 display 的大型网络的系统管理负担。其基本目标是提供了即插即显示服务，类似于终端。

display 可在任何时候被用户关闭，运行在已被关闭的 display 上的任何存在的会话必须是可鉴别的。为此，需要 3 次握手来开始一次会话。如果握手成功，任何存在的会话被立即关闭，并且开始一个新的会话。当 display 被关闭时，连接可能不被关闭。至少在 TCP 下是这样的，管理器应该通过在自己的连接上周期地进行 XSync 来减少这个问题。如果会话的连接被关闭，大概每隔 5~10 min 会话被关闭。

display 不应该因控制协议而被要求维持永久的状态，对于接收的超出序列的包的一个解决方法是在每个消息中增加消息 ID 来允许两端忽略到达超出系列的消息。为此， display 在每次启动时在最小数处增加稳定的崩溃计数，并使用这个数作为一个较大的序列数的一部分。如果 display 没有保持永久状态，则不能工作。替代的方法是管理器假定通过产生惟一的数字鉴别一个特殊会话来对永久状态负责，并且协议简单地忽略与一个无效会话相关的包。

管理器不必对包接收负责。为了防止管理器由于恶意显示引起的堵塞，没有协议要求管理器重传输一个包，这意味着任何管理器接收的有效包必须以某种方法知道阻塞 display连续重发送包。display能保持协议运行，因它总知道何时管理器已接收一个包（至少一个包的拷贝）。在管理器端，这意味着任何包可能被多次接收（如果响应是丢失的），并且复制包必须被忽略。

1.8 X Client应用程序源代码分析

X Window 系统通过 xinit 应用程序来启动，xinit 先后启动 X服务器应用程序和 X 客户端应用程序，并建立两者之间的连接。

在 Xlib 编程中，main 函数中调用函数 XOpenDisplay打开 X服务器，调用函数 XCreateWindow 创建一个窗口，然后才能调用函数 XMapWindow 显示窗口。

本节分析 xinit 应用程序中启动X 的过程，以及 Xlib 编程中常用的两个接口函数 XOpenDisplay 和 XCreateWindow，用来说明 X 协议是如何传递到传输协议中的。

1.8.1 xinit分析

在使用 startx 脚本启动 X Window 过程中，最终通过命令 xinit 来启动 X Window 系统。xinit 应用程序在分析命令行参数后，调用 startServer和 startClient 函数分别启动服务器（默认为 X）和客户端(默认为 xterm），并调用函数 XOpenDisplay 建立客户端到服务器的连接。函数 startServer 在设置环境后，运行 X服务器应用程序 server。

下面分析应用程序 xinit 的源代码，函数 main 调用层次如图 1-7所示。

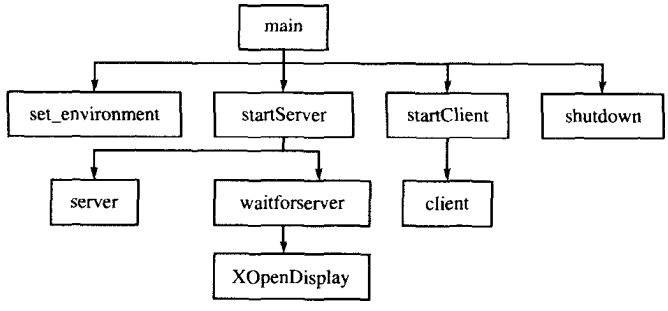


图 1-7 函数 main 调用层次

函数 main（在 xc/programs/xinit/xinit.c 中）分析如下：

```
main(int argc, char *argv[])
{
    register char **sptr = server;
    register char **cptr = client;
    register char **ptr;
    int pid;
    int client_given = 0, server_given = 0;
    int client_args_given = 0, server_args_given = 0;
    int start_of_client_args, start_of_server_args;
#ifndef X_NOT_POSIX
    struct sigaction sa;
#endif

.....
    program = *argv++;
```

```
argc--;

/* 拷贝命令行中 client相关参数 */
if (argc == 0 || (**argv != '/' && **argv != '.')) {
     for (ptr = default_client; *ptr; )
          *cptr++ = *ptr++;
} else {
     client_given = 1;
}
start_of_client_args = (cptr - client);
while (argc && strcmp(*argv, "--")) {
     client_args_given++;
     *cptr++ = *argv++;
     argc--;
}
*cptr = NULL;
if (argc) {
     argv++;
     argc--;
}

/*从命令行中拷贝 server相关参数*/
if (argc == 0 || (**argv != '/' && **argv != '.')) {
     *sptr++ = default_server;
} else {
     server_given = 1;
     *sptr++ = *argv++;
     argc--;
}
if (argc > 0 && (argv[0][0] == ':' && isdigit(argv[0][1])))
     displayNum = *argv;
else
     displayNum = *sptr++ = default_display;

start_of_server_args = (sptr - server);
while (--argc >= 0) {
     server_args_given++;
     *sptr++ = *argv++;
}
*sptr = NULL;

/*如果命令行中没有给定 client 参数，依次检查环境变量 XINITRC 和 HOME中的 XINITRC变量，得到配置文件的路径*/
if (!client_given) {
   char *cp;
   Bool required = False;

   xinitrcbuf[0] = '\0';
   if ((cp = getenv ("XINITRC")) != NULL) {
        strcpy (xinitrcbuf, cp);
        required = True;
   } else if ((cp = getenv ("HOME")) != NULL) {
        (void) sprintf (xinitrcbuf, "%s/%s", cp, XINITRC);
   }
   if (xinitrcbuf[0]) {
        if (access (xinitrcbuf, F_OK) == 0) {
           client += start_of_client_args - 1;
           client[0] = xinitrcbuf;
        } else if (required) {
           fprintf (stderr,
                 "%s:  warning, no client init file \"%s\"\n",
                 program, xinitrcbuf);
        }
   }
}
```

```
    /*如果没有给定服务器参数，检查环境变量并拷贝服务器参数到参数链表*/
    if (!server_given) {
       char *cp;
       Bool required = False;

       xserverrcbuf[0] = '\0';
       if ((cp = getenv ("XSERVERRC")) != NULL) {
            strcpy (xserverrcbuf, cp);
            required = True;
       } else if ((cp = getenv ("HOME")) != NULL) {
            (void) sprintf (xserverrcbuf, "%s/%s", cp, XSERVERRC);
       }
       if (xserverrcbuf[0]) {
            if (access (xserverrcbuf, F_OK) == 0) {
               server += start_of_server_args - 1;
               server[0] = xserverrcbuf;
            } else if (required) {
               fprintf (stderr,
                     "%s:  warning, no server init file \"%s\"\n",
                     program, xserverrcbuf);
            }
       }
    }

    //把 display 名字放到环境变量中
    set_environment ();

    /*启动 server 和 client*/
#ifdef SIGCHLD
    signal(SIGCHLD, SIG_DFL);     /* Insurance */
#endif
...
    /* 在主循环（main loop）中，让这些信号中断 wait() 调用*/
    memset(&sa, 0, sizeof sa);
    sa.sa_handler = sigCatch;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;    /* do not set SA_RESTART */

    sigaction(SIGQUIT, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGHUP, &sa, NULL);
    sigaction(SIGPIPE, &sa, NULL);

    signal(SIGALRM, sigAlarm);
    signal(SIGUSR1, sigUsr1);
    /*利用函数 Execute运行应用程序 server 和 client，在默认情况下，它们分别对应于 X 服务器应用程序和 xterm 客户端应用程序*/
    if (startServer(server) > 0
    && startClient(client) > 0) {
         pid = -1;
         while (pid != clientpid && pid != serverpid)
              pid = wait(NULL);
    }
    signal(SIGQUIT, SIG_IGN);
    signal(SIGINT, SIG_IGN);
    signal(SIGHUP, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);

    shutdown();
.....
    exit(OK_EXIT);
}
```

函数 startClient 运行客户端应用程序，列出如下：

```
static int  startClient(char *client[])
```

```
{
     if ((clientpid = vfork()) == 0) {  //子进程中
          setuid(getuid());
          setpgrp(0, getpid());
          environ = newenviron;
#ifdef __UNIXOS2__
#undef environ
          environ = newenviron;
          client[0] = (char*)__XOS2RedirRoot(client[0]); //从配置文件中得到客户端应用程序
#endif
          Execute (client,newenviron);   /运行客户端应用程序
          ......
          _exit (ERR_EXIT);
     }
     return (clientpid);
}

static int  startServer(char *server[])
{
     int old;

     old = sigblock (sigmask (SIGUSR1));

     serverpid = fork();

     switch(serverpid) {
     case 0:        // 子进程中
          sigsetmask (old);  //非阻塞

          /*
           * don't hang on read/write to control tty
           */
#ifdef SIGTTIN
          (void) signal(SIGTTIN, SIG_IGN);
#endif
#ifdef SIGTTOU
          (void) signal(SIGTTOU, SIG_IGN);
#endif
          /*在子进程中忽略 SIGUSR1 信号，在准备接受连接时，xinit 应用程序中服务器将注意信号 SIGUSR1 并回送信号 SIGUSR1*/
          (void) signal(SIGUSR1, SIG_IGN);
          /*如果客户端是 xterm -L，阻止上服务器从 vhangup() 得到 sighup。*/
          Execute (server, environ)    ;
          exit (ERR_EXIT);

          break;
     case -1:     // 出错
          break;
     default:   //父进程

          *临时修改避免 TCP 竞争，让服务器有时间在我们尝试打开 socket 之前设置其 socket 选项。使用 15s 定时还等待
SIGUSR1 信号。如果机器比 15s 慢，则可以调整这个值*/
          alarm (15);

          sigpause (old);
          alarm (0);
          sigsetmask (old);

          if (waitforserver() == 0) {  //  等待 X 服务器启动并打开 X 服务器的连接
               Error("unable to connect to X Server\r\n");
               shutdown();
               serverpid = -1;
          }
          break;
     }
```

```
    return(serverpid);
}
```

函数 waitforserver 等待 X 服务器启动，打开 X 服务器的连接，列出如下：

```
static Bool  waitforserver(void)
{
     int  ncycles    = 120;          /*循环等待的次数*/
     int  cycles;                    /*等待的循环计数*/

     for (cycles = 0; cycles < ncycles; cycles++) {
          if ((xd = XOpenDisplay(displayNum))) {
               return(TRUE);
          }
          else {
#define MSG "X Server to begin accepting connections"
              if (!processTimeout (1, MSG))
                break;
#undef MSG
          }
     }

     fprintf (stderr, "giving up.\r\n");
     return(FALSE);
}
```

1.8.2 XOpenDisplay

函数 XOpenDisplay 是 Xlib 库中的函数，用于打开一个控制显示的 X 服务器的连接。它连接到个 X 服务器，创建一个 Display 对象，返回新创建的对象的指针给调用者。该函数的原型如下：

```
Display *XOpenDisplay(display_name)
     char *display_name;
```

函数 XOpenDisplay 的参数 display_name 定义硬件 display（display 代表显示系统）的名字，它决定了 display 和使用的通信协议。如果 display_name 为空，表示使用环境变量 DISPLAY 定义的默认值。

在遵循 POSIX 的系统中，display 名字或 DISPLAY 环境变量是下面格式的字符串：

```
protocol/hostname:number.screen number
```

上面这个字符串的含义如下。

（1）protocol：定义一个协议簇。

（2）hostname 定义 display 所在主机的名字。

（3）number ：定义在这台主机上的 display 服务器的序号。

（4）screen_number：定义这台服务器使用的屏幕的序号，一台 X 服务器可以控制多个屏幕。

例如：下面的例子定义了机器名为“ dualheaded ”的 0 号 display 的 1 号屏幕。

```
dual-headed:0.1
```

函数 XOpenDisplay 返回一个 Display 结构，代表到 X 服务器的连接，其中包括所有有关 X 服务器的信息。XOpenDisplay通过 TCP 或 DECnet 通信协议或一些本地的进程间通信协议连接用户应用程序到 X 服务器。如果协议是“ tcp”、“inet ”或“inet6”，或如果没有协议被定义且 hostname 是一个主机名，并且单个“：”号分隔主机名和 display 序号，XOpenDisplay 使用 TCP流连接；如果协议被定义为“inet”，则使用 TCP over IPv4；如果协议被定义为“inet6”，则使用 TCP over IPv6；否则应用决定使用的 IP 版本。如果 XOpenDisplay 运行成功，将返回一个指向 Display 结构的指针；否则返回

NULL。在成功调用 XOpenDisplay 后，在 display 中的所有屏幕能被客户使用。

函数 XOpenDisplay（在 xc/lib/X11/OpenDis.c 中）分析如下：

```
Display *XOpenDisplay (register _Xconst char *display)
{
     ......
     /*调用连接例程得到传输连接对象，如果返回 NULL，表示连接失败。如果成功，返回连接信息*/
     if ((dpy->trans_conn = _X11TransConnectDisplay (
                          display_name, &fullname, &idisplay,
                          &iscreen, &conn_auth_name,
                          &conn_auth_namelen, &conn_auth_data,
                          &conn_auth_datalen)) == NULL) {
          Xfree ((char *) dpy);
          return(NULL);
     }

     dpy->fd = _X11TransGetConnectionNumber (dpy->trans_conn);

     /* 用连接返回的信息初始化 display 结构 dpy*/
     dpy->display_name   = fullname;
     dpy->keysyms        = (KeySym *) NULL;
     .......
     /*建立输出 buffer */
    ......
    dpy->bufmax = dpy->buffer + conn_buf_size;
     ......

/* xConnClientPrefix描述了初始的连接建立信息和授权信息*/
     ......
     prefixread = _XSendClientPrefix(dpy, &client,
                          conn_auth_name, conn_auth_data,
                          &prefix);

/*现在查看连接是否被接受*/
     //锁 display
     LockDisplay(dpy);

     if (prefixread == 0)
       /* 从 socket 中读取字节 */
       //读取 xConnSetupPrefix数据
        _XRead (dpy, (char *)&prefix,(long)SIZEOF(xConnSetupPrefix));
     ......
     _XRead (dpy, (char *)u.setup, setuplength); //读出 setup 数据

/*如果连接因故障没被服务器接受，把错误信息提供给用户 */
     if (prefix.success != xTrue) {
          ......
          return (NULL);
     }

/*授权检查成功，将相关数据加入到 display 结构中*/
     dpy->proto_major_version= prefix.majorVersion;
     dpy->proto_minor_version= prefix.minorVersion;
     ......
/*分析 setup、屏幕的各种信息*/
          //省略这部分代码

/*完成了调用内部例程，现在为外部例程解锁*/
     UnlockDisplay(dpy);
.....
/*调用同步例程，以致所有的程序能被强制同步*/
     (void) XSynchronize(dpy, _Xdebug);

/*得到请求的可利用部分和根窗口的资源管理器的数据库*/
```

```
    LockDisplay(dpy);
    {
        _XAsyncHandler async;
        _XBigReqState async_state;
         xQueryExtensionReq *qreq;
         xGetPropertyReply reply;
         xGetPropertyReq *req;
         xBigReqEnableReq *breq;
         xBigReqEnableReply brep;

         //在 dpy 的缓冲区中得到下一个可用的 X请求包 QueryExtension ,并返回它
        GetReq(QueryExtension, qreq);
        async_state.seq = dpy->request;
        async_state.opcode = 0;
        async.next = dpy->async_handlers;
        async.handler = _XBigReqHandler;
        async.data = (XPointer)&async_state;
        dpy->async_handlers = &async;
        qreq->nbytes = bignamelen;
        qreq->length += (bignamelen+3)>>2;
        Data(dpy, XBigReqExtensionName, bignamelen);

         //在 dpy 的缓冲区中得到下一个可用的 X请求包 GetProperty，并返回它
        GetReq (GetProperty, req);
        req->window = RootWindow(dpy, 0);
        req->property = XA_RESOURCE_MANAGER;
        req->type = XA_STRING;
        req->delete = False;
        req->longOffset = 0;
        req->longLength = 100000000L;

      /*等待个回答包，并拷贝它内容到特定的 reply 中同时，我们必须处理可能遇到的错误和事件包。 0表示在回答后期望的 32bit
的数据， xFalse表示不丢弃跟随“ extra ”字后面的数据*/
        if (_XReply (dpy, (xReply *) &reply, 0, xFalse)) {
             if (reply.format == 8 && reply.propertyType == XA_STRING &&
                      (reply.nItems + 1 > 0) &&
                       (reply.nItems <= req->longLength * 4) &&
                       (dpy->xdefaults = Xmalloc (reply.nItems + 1))) {
                _XReadPad (dpy, dpy->xdefaults, reply.nItems);
                dpy->xdefaults[reply.nItems] = '\0';
             }
             else if (reply.propertyType != None)
                _XEatData(dpy, reply.nItems * (reply.format >> 3));
           }

        //为 dpy 成员赋上 async 同步处理函数
        DeqAsyncHandler(dpy, &async);
        if (async_state.opcode) {
             GetReq(BigReqEnable, breq);
             breq->reqType = async_state.opcode;
             breq->brReqType = X_BigReqEnable;
             if (_XReply(dpy, (xReply *)&brep, 0, xFalse))
                dpy->bigreq_size = brep.max_request_size;
        }
    }
    UnlockDisplay(dpy);

#ifdef XKB
     XkbUseExtension(dpy,NULL,NULL);
#endif
    //成功返回
     return(dpy);
}
```

函数_X11TransConnectDisplay 尝试连接到被给出 display 名字的服务器，如果连接成功，返回文件

描述符（网络的 socket）；否则返回-1。display_name 名字格式为：

协议编号　主机名　　display 号　　屏幕号

[protocol/] [hostname] : [:] displaynumber [.screennumber]

带有两个冒号分隔主机名的字符串表示是一个 DECnet 模式名。如果一个 IPv6 地址被用做主机名，冒号会出现有 hostname 中。一个 IPv6 数字地址也可能用一个双冒号结束，因此在一行中的 3 个冒号表示 IPv6 地址用 结 束，接着是:display 。为了更容易读，一个 IPv6 数字地址主机名可能用[]括起，类似于 IETF RFC 2732 定义的 IPv6 数字地址的 URL 语法。

如果未定义主机名和协议，字符串被解释为连接到同一台机器上的服务器的本地连接，通常如下：

（1）共享内存。

（2）本地的数据流。

（3）UNIX 域 socket。

（4）连接到本地主机的 TCP。

这个函数最终调用 X 传输接口函数，X 传输接口函数期望的主机名格式如下：

```
[protocol/] [hostname] : [:] displaynumber
```

函数_X11TransConnectDisplay（在在 xc/lib/X11/ConnDis.c 中）分析如下：

```
XtransConnInfo _X11TransConnectDisplay (
    char *display_name,
    char **fullnamep,                /* RETURN */
    int *dpynump,                    /* RETURN */
    int *screenp,                    /* RETURN */
    char **auth_namep,               /* RETURN */
    int *auth_namelenp,              /* RETURN */
    char **auth_datap,               /* RETURN */
    int *auth_datalenp)              /* RETURN */
{
    .......
    /*第 0 步从参数 display_name 中查找协议、主机名、display 号、屏幕号 */
   //省略了这部分代码

    /*
     * 到了这点，就知道了下列信息：
     *    pprotocol              协议字符串或 NULL
     *    phostname              主机名字符串或 NULL
     *    idisplay               display 序号
     *    iscreen                screen 序号
     *    dnet                   是否为 DECnet 模式
     *
     * 我们现在能决定使用哪种传输协议，经分析赋给变量 pprotocol 如果 phostname 为 NULL 或 "local"， 那么选择最佳传输.
如果 phostname 为为 "unix"，那么选择 BSD UNIX 域 socket*/
   //省略这部分代码

    /*创建连接*/
    for(retry=X_CONNECTION_RETRIES; retry>=0; retry-- )
     {
        if ( (trans_conn = _X11TransOpenCOTSClient(address)) == NULL )
        {
           break;
        }
        if ((connect_stat = _X11TransConnect(trans_conn,address)) < 0 )
        {
           _X11TransClose(trans_conn);
           trans_conn = NULL;

           if (connect_stat == TRANS_TRY_CONNECT_AGAIN)
           {
                sleep(1);
```

```
            continue;
        }
        else
            break;
        }

    _X11TransGetPeerAddr(trans_conn, &family, &saddrlen, &saddr);

    /*给定的 socket 格式 (如: AF_INET)转换成 X Protocol 使用的形式 (如: FamilyInternet) */
    if( _X11TransConvertAddress(&family, &saddrlen, &saddr) < 0 )
     {
        _X11TransClose(trans_conn);
        trans_conn = NULL;
        sleep(1);
        if (saddr)
        {
            free ((char *) saddr);
            saddr = NULL;
        }
        continue;
     }

     break;
  }

 if (address != addrbuf) Xfree (address);
 address = addrbuf;

 if( trans_conn == NULL )
  goto bad;

 /*设置 close-on-exec 以至于 fork() 的程序不会混乱。 */
 _X11TransSetOption(trans_conn,TRANS_CLOSEONEXEC,1);

 return NULL;

}
```

1.8.3 函数 XCreateWindow 分析

函数 XcreateWindow 的调用层次如图 1-8 所示。

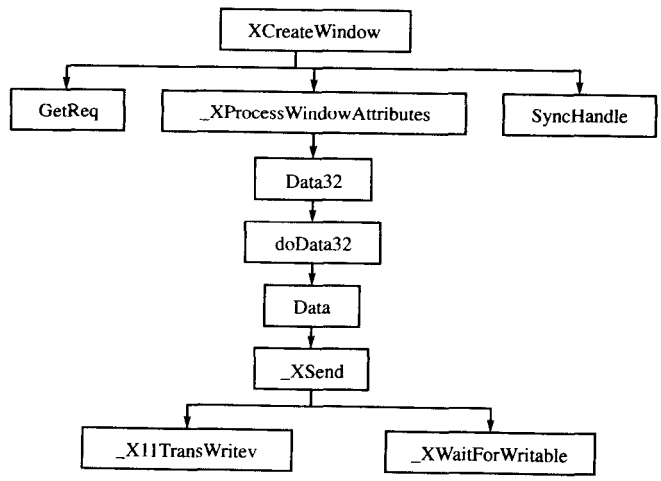


图 1-8 函数 XcreateWindow 的调用层次

该函数是 Xlib 库提供给应用程序 GUI 创建一个窗口的方法，它为一个父窗口创建了一个非映射

的子窗口。然后返回创建窗口的 ID，并引起 X 服务器产生一个 CreateNotify 事件。

函数 XCreateWindow（在 xc/lib/X11/Windows.c 中）列出如下：

```
Window XCreateWindow(dpy, parent, x, y, width, height,
            borderWidth, depth, class, visual, valuemask, attributes)
   register Display *dpy;  // 定义到 X 服务器的连接。
   Window parent;  //父窗口
   int x, y;  //窗口位置座标
   unsigned int width, height, borderWidth; //窗口的宽、高、窗口边框宽度
   int depth; //窗口 深度
   //定义创建的窗口的类，其值为 InputOutput, InputOnly 或 CopyFromParent。
   unsigned int class;
   Visual *visual; //  定义可视类型
   /*定义 attributes 参数中哪个窗口属性被定义如果 valuemask 是 0，attributes 被忽略并且不被引用 */
   unsigned long valuemask;
   /*定义值（由 valuemask 定义）被采用的结构，valuemask 值有合适的 bit 位设置指示结构中哪个属性已被设置 */
   XSetWindowAttributes *attributes;
{
   Window wid;
   register xCreateWindowReq *req;

   LockDisplay(dpy);
   /*得到在缓冲区中下一个可用的 X 请求包并返回，req 是请求指针的名字，CreateWindow 是创建窗口的请求 */
   GetReq(CreateWindow, req);
   req->parent = parent;
   req->x = x;
   req->y = y;
   req->width = width;
   req->height = height;
   req->borderWidth = borderWidth;
   req->depth = depth;
   req->class = class;
   if (visual == CopyFromParent)
   req->visual = CopyFromParent;
   else
   req->visual = visual->visualid;
   wid = req->wid = XAllocID(dpy);
   valuemask &= AllMaskBits;
   if ((req->mask = valuemask))
      _XProcessWindowAttributes (dpy, (xChangeWindowAttributesReq *)req,
               valuemask, attributes);
   UnlockDisplay(dpy);
   SyncHandle(); //使用*dpy->synchandler 同步处理函数
   return (wid);
}
```

函数 _XProcessWindowAttributes 将窗口的各种属性值放入一个数组中，并发送给 X 服务器，列出如下：

```
void _XProcessWindowAttributes (
   register Display *dpy,
   xChangeWindowAttributesReq *req,
   register unsigned long valuemask,
   register XSetWindowAttributes *attributes)
{
   unsigned long values[32];
   register unsigned long *value = values;
   unsigned int nvalues;
   /*将窗口的各种属性值放入到数组 values 中。*/
   if (valuemask & CWBackPixmap)
   *value++ = attributes->background_pixmap;

   if (valuemask & CWBackPixel)
```

```
    *value++ = attributes->background_pixel;
    ......
    req->length += (nvalues = value - values);

    nvalues <<= 2;                  /* watch out for macros... */
    Data32 (dpy, (long *) values, (long)nvalues);

}
```

宏定义 Data32（在 xc/lib/X11/Xlibint.h 中）列出如下：

```
#define Data32(dpy, data, len) _XData32(dpy, (long *)data, len)
```

函数 _XData32（在 xc/lib/X11/Xlibint.c 中）调用函数 doData32 发送数据，该函数列出如下.

```
void _XData32(
    Display *dpy,
    long *data,
    unsigned len,
{
    char packbuffer[PACKBUFFERSIZE];
    unsigned nunits = PACKBUFFERSIZE >> 2;

    for (; len > PACKBUFFERSIZE; len -= PACKBUFFERSIZE, data += nunits) {
         doData32 (dpy, data, PACKBUFFERSIZE, packbuffer);
    }
    if (len) doData32 (dpy, data, len, packbuffer);
}
```

函数 doData32 把 32 位数据放入到缓冲区中，列出如下：

```
static doData32(
    register Display *dpy
    long *data,
    unsigned len,
    char *packbuffer)
{
    long *lp,*lpack;
    long i,bits,nwords;
    long mask32 = 0x00000000ffffffff;

       lpack = (long *) packbuffer;
       lp = data;

       // nwords 是被打包为 32bit 的数量， 每个字的低 32 位将被打包成 64 位的字
       nwords = len >> 2;
       bits = 32;

       for(i=0;i<nwords;i++){
      if (bits == 32) *lpack = 0;
           *lpack ^= (*lp & mask32) << bits;
            bits = bits ^32;
            lp++;
            if(bits)
                lpack++;
       }
       Data(dpy, packbuffer, len);
}
```

函数 Data 在拷贝数据后调用 _XSend 来发送数据，列出如下：

```
void Data(
     Display *dpy,
```

```
    char *data,
    long len)
{
    if (dpy->bufptr + (len) <= dpy->bufmax) {
        memcpy(dpy->bufptr, data, (int)len);  //将数据拷贝到 data 中
        dpy->bufptr += ((len) + 3) & ~3;
    } else {
        _XSend(dpy, data, len);
    }
}
```

函数_XSend刷新 buffer 并发送客户端数据，它使用 32 位字对齐的传输，列出如下：

```
void _XSend (
    register Display *dpy,
    _Xconst char *data,
    register long size)
{
    struct iovec iov[3];
    static char const pad[3] = {0, 0, 0};
  /* XText8 和 XText16 需要填充 0 字节*/

    long skip, dbufsize, padsize, total, todo;
    _XExtension *ext;

    if (!size || (dpy->flags & XlibDisplayIOError)) return;
    dbufsize = dpy->bufptr - dpy->buffer;
    padsize = -size & 3;
    for (ext = dpy->flushes; ext; ext = ext->next_flush) { //刷新 buffer
       (*ext->before_flush)(dpy, &ext->codes, dpy->buffer, dbufsize);
       (*ext->before_flush)(dpy, &ext->codes, (char *)data, size);
       if (padsize)
         (*ext->before_flush)(dpy, &ext->codes, pad, padsize);
    }
    skip = 0;
    todo = total = dbufsize + size + padsize;

    /*有 3 个部分需要写出：display 缓冲区中的内容、用户传送的数据和需要 32 位对齐的填充数据。这个循环每次查看所有 3 个部分，
它使用 skip 指出给定部分是否是需要的*/
    while (total) {
       long before = skip;       /* amount of whole thing written */
       long remain = todo;       /*  这次尝试写出的总量  <= total */
       int i = 0;
       long len;

#define InsertIOV(pointer, length) \  //将数据放入 iovec 中
       len = (length) - before; \
       if (len > remain) \
            len = remain; \
          if (len <= 0) { \  //如果没有新的数据
               before = (-len); \
          } else { \   //把新数据放入 iovec
               iov[i].iov_len = len; \
               iov[i].iov_base = (pointer) + before; \
               i++; \
               remain -= len; \
              before = 0; \
          }

       InsertIOV (dpy->buffer, dbufsize)
       InsertIOV ((char *)data, size)
       InsertIOV ((char *)pad, padsize)

       ESET(0);
```

```
        if ((len = _X11TransWritev(dpy->trans_conn, iov, i)) >= 0) {
            skip += len;
            total -= len;
            todo = total;
        } else if (ETEST()) {
            _XWaitForWritable(dpy );
          }
         ......
    }
    dpy->last_req = (char *) & _dummy_request;
    if ((dpy->request - dpy->last_request_read) >= SEQLIMIT &&
       !(dpy->flags & XlibDisplayPrivSync)) {
       dpy->savedsynchandler = dpy->synchandler;
       dpy->synchandler = _XSeqSyncFunction;
       dpy->flags |= XlibDisplayPrivSync;
    }
    dpy->bufptr = dpy->buffer;
    return;
}
```

1.9 X Server应用程序分析

X Window 系统是高效率、高层次并与设备无关的图形显示的窗口系统，它为网络环境设计。用户可在其他的机器上运行应用程序，而在本地显示结果。 X Window 系统的客户端 X Client 与 X Server 的通信可建立在任何“可靠”的字节流上，字节流由 X协议定义。

X Server 或者说 X display Server 管理窗口和不同客户端应用程序的简单图形请求。客户端应用程序可在网络上任何机器上运行， X Server 主要完成如下 3 个方面的功能。

（1）响应对来自客户端的请求（大多数是图形和文本绘制命令）。

（2）发送设备输入（键盘和鼠标输入）和其他事件给存在的客户端。

（3）维护客户端的连接。

1.9.1 X Server概述

X Server 源代码分成 4个主要部分，其层次如图 1-9 所示。

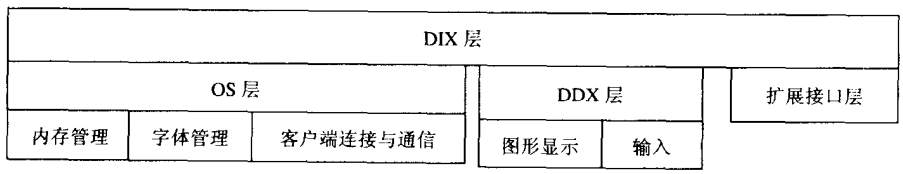


图 1-9 X Server源代码层次

X Server源代码层次说明如下。

（1）设备无关层（DIX，Device Independent）： 所有的应用共享的代码。

（2）操作系统层（OS，Operating System）： 依赖于操作系统的代码，即不同的操作系统，源代码是不同的。但在一个操作系统内，它是所有的图形设备共享的。

（3）设备依赖层（DDX，Device Dependent）：它是依赖于操作系统与图形设备的代码，是依赖于帧缓冲设备的代码。即不同的操作系统和设备，代码是不同的。

（4）扩展接口：添加新的特征到 X Server 中的标准接口。

porting layer层是对 OS 和 DDX 层组合在一起的称呼，提供接口函数。

X 资源是 Server 内部的 C 语言结构，客户应用程序按照 X 字节流协议创建并处理这些对象。客户应用程序用在网络上被传送的 32 位整数资源 ID 引用这些资源，DDX 层包括 Window（窗口）、Pixmap（位图）、Screen（屏幕）、Device（设备）、Colormap（颜色映射）、Font（字体）、Cursor（光标）和 Graphics Contexts(图形上下文)等资源,它们一般含有属性域、处理程序的指针及私有域及被 DDX 代码使用私有域保存私有数据。

这些资源结构在两个.h 中声明，一个声明资源的操作函数，另一个声明资源的结构。资源的操作函数在 DIX 层(即在 Xserver/dix 目录下的文件）中进行实现与硬件驱动无关部分，然后通过设备资源结构转接到具体设备的操作函数来执行操作。例如，在 windows.h 中声明了窗口的操作函数，在 windowstr.h 中定义了 WindowRec 结构，结构指针为*WindowOptPtr。资源的结构名通常以 Rec 结尾，结构指针以 Ptr 结尾。

1.9.2 DIX 层

DIX层代码在目录 Xserver/dix/ 下，其核心是一个分发循环。每次处理器围绕这个循环运行时，它把积累下来的输入事件从输入设备送到客户端，并且处理来自客户端的请求，这种循环是 Server 异步处理请求的最好组织办法。DIX 层中的分发循环使不同窗口之间如同有多个任务，它将每个客户端的工作分开成许多小部分。一些部分是来自客户端的请求，如图形命令；一些部分是事件分发到客户端，如用户的击键。

X Server 应用程序的主函数 main 在各种变量及结构初始化后进入 Dispatch()分发循环，当从这个分发循环退出时，表示服务器退出。此时需要清除初始化过程中分配的各种资源，然后循环进行初始化和进入 Dispatch()分发循环。

函数 main（在 Xserver/dix/main.c 中）列出如下：

```
int main(int argc, char *argv[], char *envp[])
{
    int         i, j, k, error;
    char   *xauthfile;
    HWEventQueueType alwaysCheckForInput[2];  //是 int 类型

    display = "0";

    //任何需要在服务器启动时初始化的全局变量在这个函数中初始化
    InitGlobals() ;
  ...
  //检查命令行参数
    CheckUserParameters(argc, argv, envp);
  //使用 PAM机制检查用户授权
    CheckUserAuthorization() ;

    //设置连接的限制,如最大客户端数等
    InitConnectionLimits();

    //这些变量是从某些中断处理函数中的例程所需要的，以全局变量的方式传递
    argcGlobal = argc;
    argvGlobal = argv;
    //从环境变量中查找到授权文件,它可被命令行选项重写
    xauthfile = getenv("XAUTHORITY");
    if (xauthfile) InitAuthorization (xauthfile);  //传递授权文件给全局变量
    //分析命令行参数
    ProcessCommandLine(argc, argv);

    alwaysCheckForInput[0] = 0;
    alwaysCheckForInput[1] = 1;
    while(1)  //进入个无限期的循环
```

```
    {
        serverGeneration++;  //服务器主循环计数
       //屏保的默认参数
        ScreenSaverTime = defaultScreenSaverTime;
        ScreenSaverInterval = defaultScreenSaverInterval;
        ScreenSaverBlanking = defaultScreenSaverBlanking;
        ScreenSaverAllowExposures = defaultScreenSaverAllowExposures;
#ifdef DPMSExtension  // 动态电源管理参数
        DPMSStandbyTime = defaultDPMSStandbyTime;
        DPMSSuspendTime = defaultDPMSSuspendTime;
        DPMSOffTime = defaultDPMSOffTime;
        DPMSEnabled = defaultDPMSEnabled;
        DPMSPowerLevel = 0;
#endif
        InitBlockAndWakeupHandlers();
        //执行依赖于操作系统的初始化、错误输出文件及内存检查等
        OsInit();
        if(serverGeneration == 1)  //保证第一次运行
        {
          /*创建 socket 侦听所有的客户端，重置对起始主机访问控制链表。调用函数 XdmcpInit() 初始化 XDMCP，创建 socket，计
算 display 数，建立状态机*/
            CreateWellKnownSockets();
            InitProcVectors(); //初始化客户端 ClientRec 结构指针数组
            clients = (ClientPtr *)xalloc(MAXCLIENTS * sizeof(ClientPtr));
            if (!clients)
                FatalError("couldn't create client array");

            for (i=1; i<MAXCLIENTS; i++)
                clients[i] = NullClient;

            serverClient = (ClientPtr)xalloc(sizeof(ClientRec));
            if (!serverClient)
                FatalError("couldn't create server client");
              //初始化 serverClient 对应 ClientRec 结构
            InitClient(serverClient, 0, (pointer)NULL);
        }
        else
            ResetWellKnownSockets ();

        clients[0] = serverClient;
        currentMaxClients = 1;
       //新的客户端创建时，初始化全局的资源表 clientTable
        if (!InitClientResources(serverClient))     /* for root resources */
            FatalError("couldn't init server resources");
       //设置全局的 checkForInput 数组为 alwaysCheckForInput
        SetInputCheck(&alwaysCheckForInput[0], &alwaysCheckForInput[1]);
        screenInfo.arraySize = MAXSCREENS;   //  初始化 screenInfo结构
        screenInfo.numScreens = 0;
        screenInfo.numVideoScreens = -1;

        WindowTable = (WindowPtr *)xalloc(MAXSCREENS * sizeof(WindowPtr));
        if (!WindowTable)
            FatalError("couldn't create root window table") ;


       //初始化
        InitAtoms();
        InitEvents();
        InitGlyphCaching();
        ResetClientPrivates();
        ResetScreenPrivates();
        ResetWindowPrivates();
        ResetGCPrivates();

        ResetColormapPrivates();
```

```
            ResetFontPrivateIndex();
            InitCallbackManager();
            InitVisualWrap();
            InitOutput(&screenInfo, argc, argv);
            …
            InitExtensions(argc, argv);
            if (!InitClientPrivates(serverClient))
               FatalError("failed to allocate serverClient devprivates");
            for (i = 0; i < screenInfo.numScreens; i++)
            {
               ScreenPtr pScreen = screenInfo.screens[i];
               …
            }
            InitInput(argc, argv);

            InitFonts();
            if (loadableFonts) {
               SetFontPath(0, 0, (unsigned char *)defaultFontPath, &error);
            } else {
               if (SetDefaultFontPath(defaultFontPath) != Success)
                    ErrorF("failed to set default font path '%s'",
                                defaultFontPath);
            }
            …

            for (i = 0; i < screenInfo.numScreens; i++)
               InitRootWindow(WindowTable[i]);
            DefineInitialRootWindow(WindowTable[0]);
            SaveScreens(SCREEN_SAVER_FORCER, ScreenSaverReset);
         …
            Dispatch();  //分发循环

            /* 现在释放所需要释放的资源 */

      }
      return(0);
   }
```

分发循环函数 Dispatch 在循环中处理输入事件，将输出刷新到所有的客户端。并且读取客户端请求，对客户端实现优先级调度。函数 Dispatch（在 Xserver/dix/dispatch.c 中）列出如下：

```
   void  Dispatch(void)
   {

      nextFreeClientID = 1;
      InitSelections(); //初始化 CurrentSelections 为 null，即未选中状态
      nClients = 0;

      clientReady = (int *) ALLOCATE_LOCAL(sizeof(int) * MaxClients);
      if (!clientReady)  return;

      while (!dispatchException)  //循环直到有分发错误时才退出
      {
         if (*icheck[0] != *icheck[1])
           {
              ProcessInputEvents();    //处理输入事件
            /*用输出刷新所有的客户端，然而如果一些客户端还有输入在队列（多个请求），那么不刷新，将在每次循环调度队列时阻止输出队列被刷新*/
              FlushIfCriticalOutputPending();
           }
          //等待处理请求或事件，返回等待处理的客户端数
           nready = WaitForSomething(clientReady);

   #ifdef SMART_SCHEDULE
```

```
        if (nready && !SmartScheduleDisable)
        {
           clientReady[0] = SmartScheduleClient (clientReady, nready);
           nready = 1;
        }
#endif
      /*以循环方式处理事件，每次循环处理输入*/
        while (!dispatchException && (--nready >= 0)) // 依次处理已准备好的客户端
        {
           client = clients[clientReady[nready]];
           if (! client)
            {
                /*客户端不存在，当杀死客户端( KillClient ) 时会引起这种情况发生 */
                continue;
           }
           /* GrabServer 激活能引起这种状态存在 */
           if (grabState == GrabKickout)
            {
                grabState = GrabActive;
                break;
           }
           isItTimeToYield = FALSE;

          requestingClient = client;
#ifdef SMART_SCHEDULE
            start_tick = SmartScheduleTime;
#endif
           while (!isItTimeToYield)
           {
              if (*icheck[0] != *icheck[1])
               {
                   ProcessInputEvents(); //处理输入事件
                   FlushIfCriticalOutputPending();//用输出刷新所有的客户端
               }
#ifdef SMART SCHEDULE
               if (!SmartScheduleDisable &&
                     (SmartScheduleTime - start_tick) >= SmartScheduleSlice)
               {
                   /* 惩罚浪费时间的客户端，降低优先级*/
                   if (client->smart_priority > SMART_MIN_PRIORITY)
                        client->smart_priority--;
                   break;
               }
#endif
               /* 与客户端请求打交道 */
              result = ReadRequestFromClient(client);//读取请求
              if (result <= 0)
              {//读取请求出错，关闭客户端
                   if (result < 0)
                         CloseDownClient(client);
                   break;
               }

                client->sequence++;
               if (result > (maxBigRequestSize << 2))
                  result = BadLength;
               else
                  result = (* client->requestVector[MAJOROP])(client);
               …
           }//end while (!isItTimeToYield)
           FlushAllOutput();
#ifdef SMART_SCHEDULE
           client = clients[clientReady[nready]];
           if (client)
                   client->smart_stop_tick = SmartScheduleTime;
```

```
#endif
            requestingClient = NULL;
        }          //end while (!isItTimeToYield)
        dispatchException &= ~DE_PRIORITYCHANGE;
    }             //end while (!dispatchException)
#if defined(DDXBEFORERESET)
    ddxBeforeReset ();
#endif
    KillAllClients();
    DEALLOCATE_LOCAL(clientReady);
    dispatchException &= ~DE_RESET;
}
```

1.9.3 OS层

OS层根据不同操作系统由少数例程组成，这些OS函数维护客户端的连接并调度需要为客户端所做的工作，并且提供了对字体文件的接口、对文件名翻译的字体名和底层的内存管理。函数OSInit初始化用户的OS层代码，执行log及定时器初始化等。

1. 调度和请求分发

调度和请求分发在函数dispatch中完成，其中函数WaitForSomething使Server挂起直到有来自客户端的请求或可用的输入事件或ddx注意到感兴趣的事件（图形队列准备好等）或带有缓冲的回应／事件的客户端准备好。在样例服务器上的函数WaitForSomething还有一个内建的屏幕保护，如果一段时间没有输入，会进行屏幕为全黑的屏幕保护。样例服务器是指源代码Programs/Xserver目录提供的X Server应用程序。

服务器以工作队列的机制异步处理客户端的一些操作，如doOpenFont()及do_query_bitmaps()等。

在有事件、请求和新客户端排队时，函数WaitForSomething被调用。如果多个客户端有多个请求排队，DIX将在继续处理到下一个客户端之前仅给这些请求的一部分提供服务（由isTimeToYield值来控制），因此函数WaitForSomething必须报告同一个客户端还有多少请求排队到下一次循环调度时处理。

函数WaitForSomething在调用select()之前，必须调用BlockHandle()阻塞数据处理工作，以便接收数据。通过使用函数select()来判断数据准备好的客户端，并保存这些客户端的描述符，以便进行数据处理。在函数WaitForSomething从阻塞中返回时，它必须调用例程WakeupHandler()来唤醒数据处理例程。

BlockHandle()函数调用注册在指针(* screenInfo.screens[i]->BlockHandler)(...)和(*handlers[i].BlockHandler)(...)上的具体函数进行操作，WakeupHandler()函数类似地调用注册到指针WakeupHandler上的函数。

函数WaitForSomething使用了定时器，如果在限定时间内函数操作没有完成，将强制返回。

如果在输入事件之间的时间大于屏保时间ScreenSaverTime，display被关闭（或处于屏保状态），因此WaitForSomething()必须处理屏保。更多的有关ClientsWithInput的信息，查看函数ReadRequestFromClient()。PClientsReady保存准备好的client->index值。

数据准备好的客户端的序号链表（client->index）将被保存在pClientReady中返回，序号的计数是函数调用返回值。函数Dispatch将调用函数ReadRequestFromClient()依次处理来自每个客户端的请求。

函数WaitForSomething（在Xserver/os/WaitFor.c中）分析如下：

```
int  WaitForSomething(int *pClientsReady)
{
    int i;
    struct timeval waittime, *wt;
    INT32 timeout = 0;
```

```
fd_set clientsReadable;
fd_set clientsWritable;
int curclient;
int selecterr;
int nready;
fd_set devicesReadable;
CARD32 now = 0;

FD_ZERO(&clientsReadable);

/*我们需要一个 while 循环来处理崩溃的连接和屏保超时。*/
while (1)
 {
     /*处理在工作队列中阻塞的工作*/
     if (workQueue)
        ProcessWorkQueue();
     if (XFD_ANYSET (&ClientsWithInput))
     {
         // ClientsWithInput 是在缓存中带有 FULL 请求的客户端
         XFD_COPYSET (&ClientsWithInput, &clientsReadable);
         break;
     }

     wt = NULL;
     if (timers)  //定时器时间处理
     {
         now = GetTimeInMillis();
        timeout = timers->expires - now;
        if (timeout < 0)
          timeout = 0;
        waittime.tv_sec = timeout / MILLI_PER_SECOND;
        waittime.tv_usec = (timeout % MILLI_PER_SECOND) *
                  (1000000 / MILLI_PER_SECOND);
        wt = &waittime;
     }
     XFD_COPYSET(&AllSockets, &LastSelectMask);
    /*如果 WaitForSomething()处理可能阻塞的工作,在调用 select()之前，它必须调用 BlockHandle() 。这个函数转接调用函
数(* screenInfo.screens[i]->BlockHandler)(    )和(*handlers[i].BlockHandler)(    )*/
     BlockHandler((pointer)&wt, (pointer)&LastSelectMask);
     if (NewOutputPending)
        FlushAllOutput();  //刷新输出到所有的客户端

     /*对 select()关闭这个检查以达到最小化竞争*/
     if (dispatchException)
              i = -1;
     else if (AnyClientsWriteBlocked)
     {
       //将阻塞的客户端描述符 ClientsWriteBlocked拷贝到 clientsWritable 中
        XFD_COPYSET(&ClientsWriteBlocked, &clientsWritable);
     /*函数 Select 监控等待描述符的状态变化，参数为最大描述符+1，读、写、出错描述符及超时*/
        i = Select (MaxClients, &LastSelectMask, &clientsWritable, NULL, wt);
     }
     else
     {
        i = Select (MaxClients, &LastSelectMask, NULL, NULL, wt);
     }
     selecterr = errno;
    //唤醒数据处理例程
     WakeupHandler(i, (pointer)&LastSelectMask);

     if (i <= 0) /*  函数 Select 返回错误或超时发生*/
     {
        if (dispatchException)
              return 0;
```

```
            //省略错误处理
        if (*checkForInput[0] != *checkForInput[1])
                return 0;

        if (timers)
        {
             int expired = 0;
             now = GetTimeInMillis();
             if ((int) (timers->expires - now) <= 0)
                    expired = 1;

              while (timers && (int) (timers->expires - now) <= 0)
                    DoTimer(timers, now, &timers);

          if (expired)  // 定时器超时，返回
            return 0;
        }
     }
     else
     {//函数 Select返回正常
         fd_set tmp_set;

         if (*checkForInput[0] == *checkForInput[1])
        {
            if (timers)  //  定时器时间处理，查看定时器是否超时。如果超时，则返回
            {
                int expired = 0;
                now = GetTimeInMillis();  //得到当前时间
                if ((int) (timers->expires - now) <= 0)  //已超时
                   expired = 1;

                while (timers && (int) (timers->expires - now) <= 0)
                          DoTimer(timers, now, &timers);  //   启动定时器

                if (expired) //定时器超时，返回
                       return 0;
            }
        }
         //任何写客户端被阻塞并且客户端写描述符有变化
        if (AnyClientsWriteBlocked && XFD_ANYSET (&clientsWritable))
        {
             NewOutputPending = TRUE;
             XFD_ORSET(&OutputPending, &clientsWritable, &OutputPending);
             XFD_UNSET(&ClientsWriteBlocked, &clientsWritable);
             if (! XFD_ANYSET(&ClientsWriteBlocked))
                 AnyClientsWriteBlocked = FALSE;
         }

         XFD_ANDSET(&devicesReadable, &LastSelectMask, &EnabledDevices);
         XFD_ANDSET(&clientsReadable, &LastSelectMask, &AllClients);
         XFD_ANDSET(&tmp_set, &LastSelectMask, &WellKnownConnections);
         if (XFD_ANYSET(&tmp_set))
               //将新的工作 EstablishNewConnections 加入到工作队列
               QueueWorkProc(EstablishNewConnections, NULL,
                           (pointer)&LastSelectMask);

         if (XFD_ANYSET (&devicesReadable) || XFD_ANYSET (&clientsReadable))
                           break;
     }//end函数 Select返回正常
} //end while (1)
 //得到数据准备好的客户端的序号链表及序号数量用来返回
nready = 0;
if (XFD_ANYSET (&clientsReadable))
{
    for (i=0; i<howmany(XFD_SETSIZE, NFDBITS); i++)
```

```
        {
            int highest_priority = 0;

            while (clientsReadable.fds_bits[i])
            {
                int client_priority, client_index;

                curclient = ffs (clientsReadable.fds_bits[i]) - 1;
                client_index = /* raphael: modified */
                    ConnectionTranslation[curclient + (i * (sizeof(fd_mask) * 8))];

                pClientsReady[nready++] = client_index;
                clientsReadable.fds_bits[i] &= ~(((fd_mask)1L) << curclient);
            }
        }
    }
    return nready;
}
```

2. 新建到客户端的连接

新建到客户端的连接，该连接依赖于用户的字节流机制，如 TCP/IP 等。shell 环境变量 DISPLAY 的值（如 testhost:0.0）定义了主机为 testhost，第 1 个 0 表示有多个 X Server 进程中的第 0 号 display。每个运行的 X Server 进程被称为“display”，第 2 个 0 表示第 0 号屏幕。

例如，一台工作站 fredsbittube 通过 TCP/IP 在远程超级计算机 Cray 上运行一个应用程序，工作站上的环境变量 DISPLAY=fredsbittube:0.0。工作站是 X Server，Cray 是 X Client 客户端调用 TCP 例程请求 X 建立连接，它查找服务器的 IP 地址，并发出一个打开请求给服务器端口号为 6000 的 TCP 端口。

所有的 X 服务器侦听 6000 端口，并检查请求的主机是否在服务器的 trustworthy 链表中。如果是，服务器打开另一个端口与客户端通信。

函数 EstablishNewConnections 建立服务器到新客户端的连接，如果任何客户端正在侦听 socket，接受它们，返回新客户端的序号的掩码并更新 AllClients 和 AllSockets。

函数 EstablishNewConnections（在 Xserver/os/connection.c 中）列出如下：

```
Bool  EstablishNewConnections(ClientPtr clientUnused, pointer closure)
{
    fd_set  readyconnections;  //准备好的 listener 集
    int curconn;               //当前准备好的 listener 的 fd
    register int newconn;      //新的客户端的 fd
    CARD32 connect_time;
    register int i;
    register ClientPtr client;
    register OsCommPtr oc;
    fd_set tmask;

    XFD_ANDSET (&tmask, (fd_set*)closure, &WellKnownConnections);
    XFD_COPYSET(&tmask, &readyconnections);
    if (!XFD_ANYSET(&readyconnections))
                return TRUE;
    connect_time = GetTimeInMillis();
    /*杀死出错或无效的客户端 */
    for (i=1; i<currentMaxClients; i++)
    {
        if ((client = clients[i]))
        {
            oc = (OsCommPtr)(client->osPrivate);
            if ((oc && (oc->conn_time != 0) &&
                (connect_time - oc->conn_time) >= TimeOutValue) ||
                (client->noClientException != Success && !client->clientGone))
                        CloseDownClient(client);
        }
```

```
    }
    //调用传输层的函数接受连接
    for (i = 0; i < howmany(XFD_SETSIZE, NFDBITS); i++)
    {
       while (readyconnections.fds_bits[i])
       {
            XtransConnInfo trans_conn, new_trans_conn;
            int status;

            curconn = ffs (readyconnections.fds_bits[i]) - 1;
            readyconnections.fds_bits[i] &= ~((fd_mask)1 << curconn);
            curconn += (i * (sizeof(fd_mask)*8));

            if ((trans_conn = lookup_trans_conn (curconn)) == NULL)
               continue;

            if ((new_trans_conn = _XSERVTransAccept (trans_conn, &status)) == NULL)
                continue;

            newconn = _XSERVTransGetConnectionNumber (new_trans_conn);

            if (newconn < lastfdesc)
            {
                 int clientid;
                 clientid = ConnectionTranslation[newconn];
                 if(clientid && (client = clients[clientid]))
                        CloseDownClient(client);
            }

            _XSERVTransSetOption(new_trans_conn, TRANS_NONBLOCKING, 1);

            if (!AllocNewConnection (new_trans_conn, newconn, connect_time))
            {
               ErrorConnMax(new_trans_conn);
               _XSERVTransClose(new_trans_conn);
            }
       }
    }
    return TRUE;
}
```

3. 从客户端读取数据

使用函数 ReadRequestFrom 从客户端读取数据，该函数通过传输层函数读取数据。函数 ReadRequestFrom（在 Xserver/os/io.c 中）分析如下：

```
int  ReadRequestFromClient(ClientPtr client)
{
   OsCommPtr oc = (OsCommPtr)client->osPrivate;
   ConnectionInputPtr oci = oc->input;
   int fd = oc->fd;
   unsigned int gotnow, needed;
   int result;
   register xReq *request;
   Bool need_header;
```

/* 如果个输入 缓存是空，或者缓存太大或被链接到空输入缓存链表中，则释放它。这意味着不同的客户端可以在不同时刻共享同一输入缓存，这样处理可节约内存 */

```
    if (AvailableInput) // OsCommRec 结构指针 AvailableInput 不为空
    {
         if (AvailableInput != oc)
         {
            register ConnectionInputPtr aci = AvailableInput->input;
            if (aci->size > BUFWATERMARK)
```

```
        {
            xfree(aci->buffer);
            xfree(aci);
        }
        else
        {
            aci->next = FreeInputs;
            FreeInputs = aci;
        }
        AvailableInput->input = (ConnectionInputPtr)NULL;
    }
    AvailableInput = (OsCommPtr)NULL;
}

/*确保有一个输入缓存 */
if (!oci)  //ConnectionInput 结构指针 oci 为空
{
    if ((oci = FreeInputs))
    {
        FreeInputs = oci->next;
    }
    else if (!(oci = AllocateInputBuffer()))
    {
        YieldControlDeath();
        return -1;
    }
    oc->input = oci;
}

/*先开始下一个请求 */
oci->bufptr += oci->lenLastReq;
need_header = FALSE;
gotnow = oci->bufcnt + oci->buffer - oci->bufptr;
if (gotnow < sizeof(xReq)) // 缓存尺寸小于 xReq 的大小
 {
    /* 我们还没有一个整个 xReq，不能区分请求的大小直到得到整个*/
    needed = sizeof(xReq);
    need_header = TRUE;
}
else
{
    /* 我们有一个整个 xReq，可能区分整个请求的大小，除非它是一个 Big Request*/
    request = (xReq *)oci->bufptr;
    needed = get_req_len(request, client);
    client->req_len = needed;
    needed <<= 2;    // 得到 byte 形式
}

if (gotnow < needed)
{
    /*需要读更多的数据，以便得到整个 xReq（如果 need_header 为 TRUE），或整个 xBigReq（如果 move_header 为 TRUE，
或请求的剩余部分（如果 need_header和 move header都为 FALSE）*/
    oci->lenLastReq = 0;
    if (needed > MAXBUFSIZE)
    {
        /*请求太大以致于不能处理*/
        YieldControlDeath();
        return -1;
    }

    if ((gotnow == 0) || ((oci->bufptr - oci->buffer + needed) > oci->size))
    {
        /* 没有数据或请求太大与缓存不匹配 */
        if ((gotnow > 0) && (oci->bufptr != oci->buffer))
            /*存储我们已读取的数据*/
```

```
        memmove(oci->buffer, oci->bufptr, gotnow);
    if (needed > oci->size)
    {
        /*增大缓存以便容纳请求*/
        char *ibuf;
        ibuf = (char *)xrealloc(oci->buffer, needed) ;
        if (!ibuf)
         //缓存分配失败
           YieldControlDeath();
           return -1;
        }
        oci->size = needed;
        oci->buffer = ibuf;
    }
    oci->bufptr = oci->buffer;
    oci->bufcnt = gotnow;
}
/* 这个函数在 trans conn被释放后调用，在这种情况下 trans_conn 将为 null，这种情况应该不会发生。*/
if (!oc->trans_conn)
{
    /*如同在读时发生的错误一样对待。 */
    YieldControlDeath();
    return -1;
}
//调用传输层的函数读取数据
result = _XSERVTransRead(oc->trans_conn, oci->buffer + oci->bufcnt
                oci->size - oci->bufcnt) ;
if (result <= 0)
 //读取数据错误
    if ((result < 0) && ETEST(errno))
    {
        YieldControlNoInput();
        return 0;
    }
    YieldControlDeath();
    return -1;
}
oci->bufcnt += result;
gotnow += result;
/*在大请求后释放一些空间 */
if ((oci->size > BUFWATERMARK) &&
          (oci->bufcnt < BUFSIZE) && (needed < BUFSIZE))
{
    char *ibuf;

    ibuf = (char *)xrealloc(oci-  >buffer, BUFSIZE);  // 分配缓存
    if (ibuf)  //缓存分配成功
    {
        oci->size = BUFSIZE;
        oci->buffer = ibuf;
        oci->bufptr = ibuf + oci->bufcnt - gotnow;
    }
}

if (need_header && gotnow >= needed)
{
    /* 我们需要一个 xReq，现在已得到它。*/
    request = (xReq *)oci->bufptr;
    needed = get_req_len(request, client);
    client->req_len = needed;
    needed <<= 2;
}

if (gotnow < needed)
{
```

```
            /* gotnow还不够时，返回。 */
            YieldControlNoInput();
            return 0;
        }
    }//end if (gotnow < needed)

    if (needed == 0)
    {
        needed = sizeof(xReq);
    }
    oci->lenLastReq = needed;
```

/*检查客户端返回给调用者的请求是否至少个整个请求在缓存中如果仅有一个部分请求，则将缓存作为空对待，以便 select() 再次得到调用并且其他客户端能进入队列*/

```
    gotnow -= needed;
    if (gotnow >= sizeof(xReq))
    {
        request = (xReq *)(oci->bufptr + needed);
        if (gotnow >= (result = (get_req_len(request, client) << 2)) )
            FD_SET(fd, &ClientsWithInput);
        else
        {
            YieldControlNoInput();
        }
    }
    else
    {
        if (!gotnow)        AvailableInput = oc;
        YieldControlNoInput();
    }

    if (++timesThisConnection >= MAX_TIMES_PER)  //连接超时
            YieldControl();
    client->requestBuffer = (pointer)oci->bufptr;
    return needed;
}
```

4. 发送事件、错误并回应客户端

函数 WriteToClient 将缓存区中的数据字节写回指定的客户端。

函数 SendErrorToClient 返回错误信息给客户端。

函数 FlushAllOutput、FlushIfCriticalOutputPending 和 SetCriticalOutputPending 支持对客户端的缓存写或延迟写操作，函数 FlushAllOutput 无条件地刷新所有的输出到客户端，函数 FlushIfCriticalOutputPending 仅在 SetCriticalOutputPending 被调用时刷新输出到客户端。

这些函数最终调用传输层的函数来完成操作。

5. 字体支持

在样例服务器中，字体在硬盘上编码或从字体服务器中获得。对于硬盘字体，每种字体有一个文件，文件名类似 fixed.pcf，字体服务器字体通过 X 字体服务器协议从网络中读取。装有硬盘字体的硬盘目录和字体服务器的名字目前放在 font path 中，字体接口的代码在 ../fonts/lib 下。

6. 内存管理

内存管理基于 C 运行库的函数，Xalloc()、Xrealloc() 和 Xfree()的工作几乎与 malloc()、realloc()和 free()一样。除了能传 null 指针到 Xrealloc()和 Xfree()中之外，传输 null 指针到 Xrealloc()时，Xrealloc()分配个新的内存，传输到 Xfree()时对其无影响。

7. 客户端调度

X Server 有能力用类似于操作系统的调度方法调度客户端，挂起或重启动客户端而不考虑其输入缓存的状态。这个功能允许 X Server 在等待长时间的网络动作（如装载字体）时挂起一个客户端，并在继续处理第 1 个客户端之间处理另一个客户端的请求。当两个客户端同时绘制其窗口时，调度算法对执行效率有很大的影响。

通过设置全局变量 isItTimeToYield 可以告诉 DIX 客户端的“时间片”的结束，并留心下一个客户端。在当前的请求完成后，DIX 将移到下一个客户端。在样例服务器中，函数 ReadRequestFromClient() 在 10 个请求包被从同一个客户端读取后设置 isItTimeToYield。

调度函数有 IgnoreClient、AttendClient、ClientSleep、ClientSignal 和 ClientWakeup 等。

8. 其他 OS 函数

其他错误信息打印函数用来打印从 dix 和 ddx 层输出的诊断信息。

9. 习惯语法的支持

DBE 规范介绍了习惯语法的概念，X 请求成组作为整体执行比单个序列执行更有效率， X Server 内部支持 DBE 规范定义的习惯语法。

1.9.4 DDX层

DDX 层通过不同结构指向的例程完成大部分工作，这些资源的行为由这些例程指针决定。大多数例程用于屏幕上的图形显示或对图形显示的支持，其他功能如处理从输入设备的用户输入等。

X 输入设备的几种类型是键盘、指针设备和其他设备。X Server 还支持在《X 输入扩展》规范中描述的扩展设备。核心设备实际上被应用做两类设备集合，其中鼠标是一个 ButtonDevice、一个 ValuatorDevice 和一个 PtrFeedbackDevice；键盘是一个 KeyDevice、一个 FocusDevice 和一个 KbdFeedbackDevice。从 DIX 层看，输入设备的细节被隐藏。

1. 输入设备数据结构

DIX 在被称为“InputInfo”的数据结构中保存了全局的设备目录，每台设备有一个设备结构 DeviceRec，DIX 能通过 InputInfo来定位任何 DeviceRec 结构。另外，它还有一个特殊的指向主指针设备 ID 的指针和一个指向主键盘 ID 的指针。DeviceRec（Xserver/include/input.h ）是一个设备无关结构，其中保存一台输入设备的状态。 DevicePtr 是指向 DeviceRec 的指针。

一个 xEvent 结构描述了服务器将报告给一个客户端的事件，定义在 Xproto.h 中，它是一个有所有种类事件的联合结构。

结构 DMXInputInfo 在 DMX 层描述了输入设备的抽象属性，还提供了输入设备所共有的输入处理例程指针等。通过该结构，DMX 层可以查找到具体的输入设备，并调用具体输入设备的各种例程。

结构 DMXInputInfo（在 Xserver/hw/dmx/dmxinput.h 中）分析如下：

```
extern DMXInputInfo        *dmxInputs;   // 输入链表
typedef struct _DMXInputInfo DMXInputInfo;
/* DMXInputInfo 是在 dmx.h 中声明的类型，所有的函数可以访问这个全局的指针但成员仅在输入相关例程是可用的 */
struct _DMXInputInfo {
    const char           *name; // 输入显示或设备的名字，来自命令行或配置文件
    Bool                 freename; //  如果为 true，在销毁时释放 name
    Bool                   detached; //如果为 true，输入屏幕是分离的
    int                  inputIdx; //dmxInput 的全局的序号
    int                  scrnIdx;  // dmxScreens 的全局的序号
    Bool                   core;  //如果为 true，初始这些设备作为发送核心事件的设备
    Bool                   console; //如果控制台和后端输入共享同一后端 display 时，它为 true
```

```
    Bool                windows; //如果窗口轮廓被画在控制台上，它为 true

    ProcessInputEventsProc  processInputEvents;  //处理输入事件的例程
    UpdateWindowInfoProc    updateWindowInfo;  //  更新窗口信息的例程

                            /*本地的输入信息*/
    dmxSigioState           sigioState;    //当前状态
    int                     sigioFdCount;  //正在使用的 fd 数量
    int                     sigioFd[DMX_MAX_SIGIO_FDS];    //fd的链表
    Bool                    sigioAdded[DMX_MAX_SIGIO_FDS]; //  激活的 fd

    int                 vt_switch_pending;  //  如果一个 VT 正切换到挂起但还没挂起，则为 true
    int                 vt_switched;    //如果 VT 切换已发生，则为 true
    int                 numDevs;    //被处理的设备数量

    /*实际的输入设备链表，每个_DMXInputInfo   结构能引用多于一台设备。如：键盘和后端 display 的指针，或所有的在后端 display
上的 XInput 扩展设备。*/
    DMXLocalInputInfoPtr    *devs;  //  包含输入设备操作的各种函数指针，提供给低层输入例程用

    char                *keycodes; //来自命令行的键盘的键编码
    char                *symbols;  //来自命令行的键盘的符号
    char                *geometry; //来自命令行的键盘排列
};
```

结构 InputInfo 以分类的形式链接了所有的输入设备，列出如下：

```
extern InputInfo inputInfo;
typedef struct {
    int             numDevices;     //设备的总数
    DeviceIntPtr    devices;        //启动的所有设备
    DeviceIntPtr    off_devices;    //关闭的所有设备
    DeviceIntPtr    keyboard;       //服务器主键盘
    DeviceIntPtr    pointer;        //指针设备，如鼠标
} InputInfo;
```

结构_DeviceIntRec 描述了每个输入设备的属性和例程等，分析如下：

```
typedef struct _DeviceIntRec {
    DeviceRec  public;  //  输入设备共有属性和例程
    DeviceIntPtr next;    //  下一台输入设备
    TimeStamp  grabTime;
    Bool   startup;       // 在服务器初始化时需要被启动时为 true
    DeviceProc deviceProc;    /   / proc(DevicePtr, DEVICE_xx).     它被用来初始化、启动或关闭这台设备
    Bool   inited;                //   如果初始化成功时，则为 true
    GrabPtr    grab;              //DIX    使用的 grabber
    struct {
         Bool       frozen;
         int        state;
         GrabPtr          other;          //如果其他的 grab 捕获住设备，用这个指针指向其他的 grab
         xEvent     *event;             //事件，存储以便再次使用或回溯
         int        evcount;
    } sync;
    Atom        type;
    char        *name;
    CARD8            id;
    CARD8            activatingKey;
    Bool        fromPassiveGrab;
    GrabRec          activeGrab;
    void        (*ActivateGrab) (
                DeviceIntPtr /*device*/,
                GrabPtr /*grab*/,
                TimeStamp /*time*/,
                Bool /*autoGrab*/);
    void        (*DeactivateGrab)(
                DeviceIntPtr /*device*/);
```

```
    KeyClassPtr      key;    // 键相关信息，如键编码、符号及状态等
    //指针设备的/移到处理函数指针、移动所在窗口、指针所在点分辨率信息及模式等
    ValuatorClassPtr valuator;
    ButtonClassPtr   button; // 鼠标按钮信息，包括按钮状态、按钮序号、移动掩码及映射值等
    FocusClassPtr    focus;  //焦点信息，包括所在窗口、时间戳及轨迹大小等
    ProximityClassPtr    proximity;  // 邻近的字符或填充字符
    KbdFeedbackPtr   kbdfeed;  //键盘响铃、控制等例程的指针
    PtrFeedbackPtr   ptrfeed;      指针设备的控制、处理等例程的指针
    IntegerFeedbackPtr   intfeed;  //整合在一起的设备控制、处理等例程的指针
    StringFeedbackPtr    stringfeed;   / 字符串的控制、处理等例程的指针
    BellFeedbackPtr bell;   //响铃的控制、处理等例程的指针
    LedFeedbackPtr   leds;   //led指示灯的控制、处理等例程的指针
#ifdef XKB
    struct _XkbInterest *xkb_interest;  //X 下键盘相关信息
#endif
    DevUnion         *devPrivates;  //绘画环境相关信息
    int             nPrivates;
    DeviceUnwrapProc    unwrapProc;  //设备处理例程
} DeviceIntRec;
```

结构_DMXInputInfo 在 DMX 层描述了输入设备的输入事件例程及输入设备的链表*devs 等，通过链表*devs 可查找到设备在 DMX 层的信息结构 DMXLocalInputInfoRec，在这个结构中可找到设备的具体描述结构 DeviceIntRec。

结构 InputInfo 描述了所有输入设备，它将输入设备按类型组成链表；结构 DeviceIntRec 抽象化输入设备，它包括了输入设备的各种参数及操作函数指针；结构 DeviceRec 描述了输入设备的共有特征及操作函数。

输入设备的结构关系如图 1-10 所示。由于输入设备结构很复杂，所以其中省略了 DeviceIntRec 结构中许多指针各种属性的结构，而只列出了这些结构的名字。

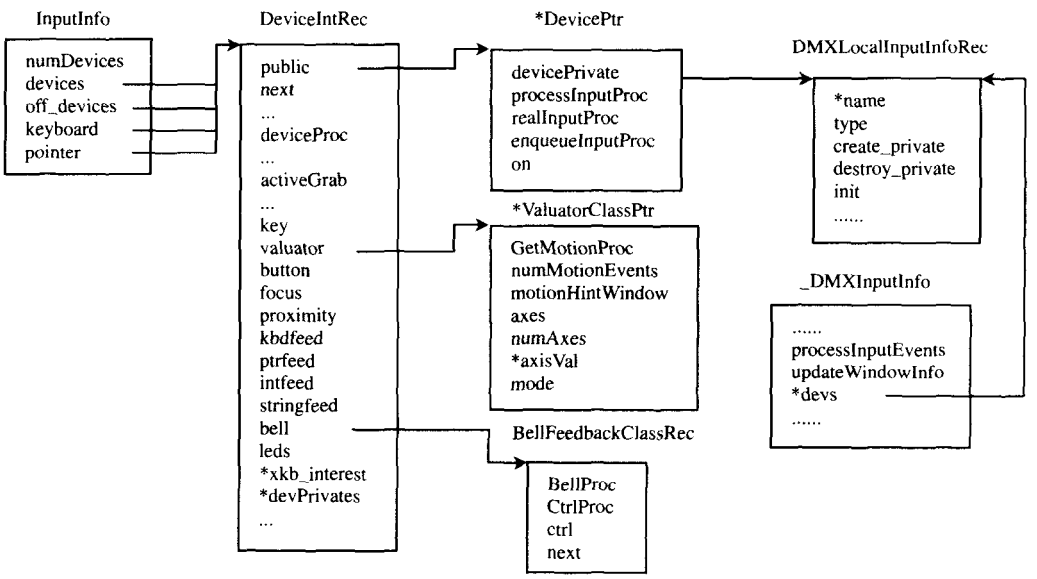


图 1-10　输入设备的结构关系

2. 处理事件

主要的 DDX 输入接口是例程 ProcessInputEvents()，这个例程分发输入事件。当输入被挂起，甚至于在未挂起时，DIX 调用它。ProcessInputEvents() 从每台设备得到事件并分发事件到 DIX 层。为了分发事件到 DIX 层，DDX 应该调用下面的例程：

```
void DevicePtr->processInputProc(pEvent, device, count)
xEventPtr events;
DeviceIntPtr device;
int count;
```

这是设备的“input proc”（输入程序），它是一个 DIX 程序。DIX 将在第 1 次调用 ProcessInputEvents() 时填充这个程序指针到其自己的一个例程中，并处理应被分发的多个事件。多少个“input proc”例程需要被调用多少次，DIX 将事件缓存起来并发送它们。

例如，用户的 ProcessInputEvents() 例程可能检查鼠标和键盘。如果有几次击键事件排队，则能调用几次键盘的 processInputProc 例程，并刷新内部队列。

事件是传递到“ input proc ”中的 xEvent 结构。

无论输入事件何时要求被处理，函数 ProcessInputEvents 都从 dix/dispatch.c 的函数 Dispatch() 中调用，所有的工作在底层的输入事件处理例程上完成。

函数 ProcessInputEvents（在 Xserver/hw/dmx/dmxinput.c 中）分析如下：

```
void ProcessInputEvents(void)
{
    int         i;
    DMXInputInfo *dmxInput;
    //调用 DMXInputInfo 结构成员 processInputEvents 指针所指向的输入事件处理函数
    for (i = 0, dmxInput = &dmxInputs[0]; i < dmxNumInputs; i++, dmxInput++)
        if (!dmxInput->detached && dmxInput->processInputEvents)
            dmxInput->processInputEvents(dmxInput);
}
```

3. 告诉 DIX何时输入被挂起

在服务器的分发循环中，无论 WaitForSomething() 何时返回，DIX 检查看是否有任何正在挂起的输入设备。

检查输入挂起必须非常快，DIX 比较两个指针所指的值*icheck[0]和*icheck[1]是否相同。如果不同，那么假定输入正在挂起， DIX 调用 ProcessInputEvents()。 函数 SetInputCheck(c0, c1) 为这两个指针赋值，开发人员应该在输入设备初始化时调用这个函数。

4. 控制输入设备

例程 GetMotionProc 为客户端报告移动事件，并返回移动事件的数量。当用户拖动指针设备时，光标位置扫过有限个点。通常，与这些点相关，而不是与开始点和结束点相关的客户端将接收指针移动事件。只要服务器产生这些移动事件，则移动事件不排队。在队列中，每个新的事件代替上一个事件。如果需要，一个服务器可以存储这些中间的事件在移动缓存中。然后一个客户端，如绘画程序，可以请求将这些事件使用 GetMotionProc 例程分发给它。

可以通过 BellProc（percent, pDevice, ctrl, unknown）例程来控制键盘上的铃声，声音从小到大为 0~100 级。

例程 CtrlProc（device, ctrl）为指针设备或键盘设备提供了控制，当一个客户端已请求为特定设备改变当前的设置时，DIX 层调用它来通知 DDX 层。对于指针设备来说，这可能是一个位置报告的放大因子。

5. 输入初始化

输入设备包括键盘及指针设备（如鼠标），输入设备在初始化后，具体设备的各种处理例程被添加到输入设备结构的对应指针上。这样，通过输入设备结构即可控制设备。

输入设备初始化通过函数 InitInput进行,它调用层 DDX层的函数 dmxAddDevice 来注册输入设备，创建设备对象并加入到输入设备链表中。

函数 InitInput 的调用层次如图 1-11 所示。

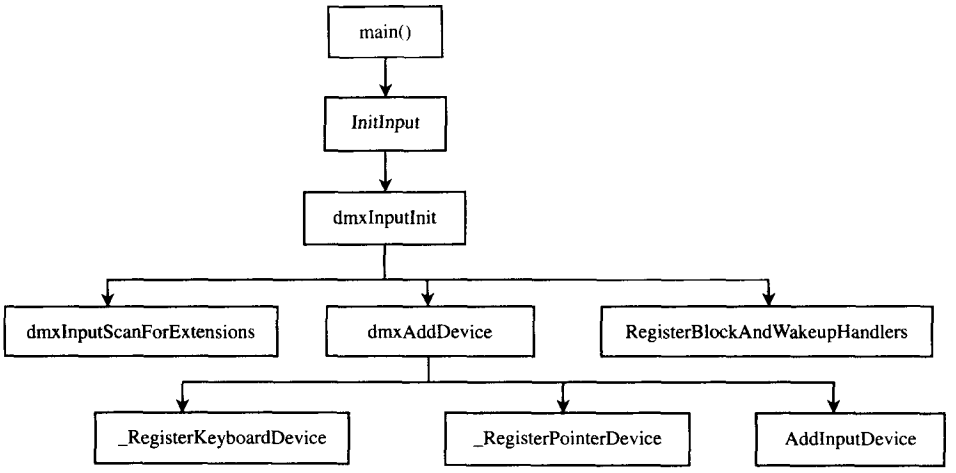


图 1-11　函数 InitInput 的调用层次

函数 AddInputDevice() 为输入设备赋上设备处理例程给输入设备结构中的 DeviceProc 指针。在指针 DeviceProc(device, action) 中，根据 action 的不同枚举类型设置不同的处理例程，分别说明如下（在 input.h 中）。

（1）DEVICE INIT：如果 action 为 DEVICE_INIT，表示设备初始化。设备调用 InitPointerDevice Struct()和 InitKeyboardDeviceStruct() ，或类似的例程初始化设备。如果返回非零值，表示设备初始化失败。

（2）DEVICE ON：如果 action 为 DEVICE ON ，表示允许开始通过设备结构中的 ProcessInputProc 例程将事件放入客户端数据流中。

（3）DEVICE OFF：如果 action 为 DEVICE OFF ，表示设备没有事件给 DIX 层，设备将以僵死的状态报告给用户。

（4）DEVICE CLOSE：如果 action 为 DEVICE CLOSE （在设备关闭或重启），表示设备被关闭。

下面按照图 1-11 分析输入设备的初始化函数，为了直观，只列出了与设备初始化相关代码。

函数 InitInput 执行输入设备的初始化，列出如下：

```
void InitInput(int argc, char **argv)
{
   int        i;
   DMXInputInfo *dmxInput;
   ...
   for (i = 0, dmxInput = &dmxInputs[0]; i < dmxNumInputs; i++, dmxInput++)
      dmxInputInit(dmxInput);
   ...
}
```

函数 dmxInputInit 初始化在参数 dmxInput 中描述的设备，列出如下：

```
void dmxInputInit(DMXInputInfo *dmxInput)
{

    /*定位 X display 的键盘或指针设备扩展*/
   dmxInputScanForExtensions(dmxInput, doXI);

   for (i = 0; i < dmxInput->numDevs; i++) {
      DMXLocalInputInfoPtr dmxLocal = dmxInput->devs[i];
```

```
        dmxLocal->pDevice = dmxAddDevice(dmxLocal);
        if (dmxLocal->isCore) {
            if (dmxLocal->type == DMX_LOCAL_MOUSE)
                pPointer  = dmxLocal->pDevice;
            if (dmxLocal->type == DMX_LOCAL_KEYBOARD)
                pKeyboard = dmxLocal->pDevice;
        }
    }

    //例程 dmxProcessInputEvents从队列 dmxEventQueue 中移出事件并处理
    dmxInput->processInputEvents    = dmxProcessInputEvents;
    dmxInput->detached              = False;
    //注册阻塞与唤醒函数
    RegisterBlockAndWakeupHandlers(dmxBlockHandler
                                   dmxWakeupHandler
                                   (void *)dmxInput->inputIdx);
}
```

函数 dmxAddDevice 注册键盘或指针设备，列出如下：

```
static DeviceIntPtr dmxAddDevice(DMXLocalInputInfoPtr dmxLocal)
{
   .....
    dmxInput = &dmxInputs[dmxLocal->inputIdx];

    //赋注册例程到 registerProcPtr指针
    if (dmxLocal->sendsCore) {
        if (dmxLocal->type == DMX_LOCAL_KEYBOARD && !dmxLocalCoreKeyboard) {
            dmxLocal->isCore     = 1;
            dmxLocalCoreKeyboard = dmxLocal;
            name                 = "keyboard";
            registerProcPtr      = _RegisterKeyboardDevice;  //键盘设备注册例程
        }
        if (dmxLocal->type == DMX_LOCAL_MOUSE && !dmxLocalCorePointer) {
            dmxLocal->isCore     = 1;
            dmxLocalCorePointer  = dmxLocal;
            name                 = "pointer";
            registerProcPtr      = _RegisterPointerDevice;  //指针设备注册例程
        }
    }
    ......
    pDevice                   = AddInputDevice(dmxDeviceOnOff, TRUE);
   ......
    pDevice->public.devicePrivate = dmxLocal;
    dmxLocal->pDevice             = pDevice;

    devname      = dmxMakeUniqueDeviceName(dmxLocal);
    atom         = MakeAtom((char *)devname, strlen(devname), TRUE);
    pDevice->type = atom;
    pDevice->name = devname;

    registerProcPtr(pDevice); //使用具体的注册例程注册键盘或指针设备

    if (dmxLocal->isCore && dmxLocal->type == DMX_LOCAL_MOUSE)
        miRegisterPointerDevice(screenInfo.screens[0], pDevice);

    if (dmxLocal->create_private)
        dmxLocal->private = dmxLocal->create_private(pDevice);
    return pDevice;
}
```

函数_RegisterPointerDevice 为指针设备注册各种操作例程，列出如下：

```
void _RegisterPointerDevice(device)
    DeviceIntPtr device;
{
```

```
    inputInfo.pointer = device;
    device->public.processInputProc = ProcessPointerEvent;
    device->public.realInputProc = ProcessPointerEvent;

    device->ActivateGrab = ActivatePointerGrab;
    device->DeactivateGrab = DeactivatePointerGrab;
    if (!device->name)
    {
        char *p = "pointer";
        device->name = (char *)xalloc(strlen(p) + 1);
        strcpy(device->name, p);
    }
}
```

6. 键盘映射及键编码

协议定义了键编码值范围为 8~255，一旦在 DIX 层建立了键值映射，DIX 保护并维护其在客户端的变化。X 协议定义了标准名字指定在每个键上的名字（见 x11/keysym.h），如果一个客户端尝试修改一个键（如用“ A ”键作为“ Control ”键）. DIX 调用 LegalModifier 例程。如果指定设备上的键被修改成功，返回 TRUE。

1.9.5 Screen

屏幕显示的硬件要求仅是能从帧缓冲中读写数据，所有的显示应支持 32 位或少于 32 位的颜色深度。X 使用像素作为屏幕上的距离单位。

1. 分布式多头 X 设计

分布式多头 X 设 计（Distributed Multihead X design，DMX ）通过开发一个控制多个后端 X Server 的前端代理服务器组成大的 display 来实现。

X Server 将扮演一套后端 X Server的一个代理，后端 X Server 处理所有的可视对象。 X Client 将如同正常连接一样连接前端 X Server 前端服务器将呈现一个抽象的视图给单个 display 的客户端，从而确保标准的 X Client 将在没有修改的情况下继续操作（通过视图的限制和 X Server 提供的扩展）。客户端能使用可获得后端服务器信息的扩展（如对于弹出窗口的放置，通过窗口管理器进行窗口排列等）。DMX 服务器的框架分为两个主要的部分，即输入（如鼠标和键盘）和输出（如图形显示和窗口请求）。

DMX 服务器能从 3 类输入设备接收输入，即“ 本地 ”设备（物理上依附于 DMX运行的机器）、“ 后端 ”设备（物理上依附于一个或多个后端 X 服务器，并且从后端通过 X 协议产生事件。）和“ 控制台 ”设备（可从任何非后端 X 服务器抽象）。后端和控制台设备被不同看待，因为在后端 X服务器上的指针设备也能控制硬件 X 光标的位置。

图形请求将被前端服务器接受，可见窗口的图形绘制将按需要分开，并通过 X11 库被送到合适的后端 X服务器完成图形绘制，基本的框架工作将遵循 X 套接模式访问。GC 将在前端服务器中被管理，并按请求被发送到合适的后端服务器。位图绘制将被前端 X 服务器处理，窗口请求（如排序、映射及移动等）将在前端服务器中处理。如果请求需要一个视图变化，窗口操作将被转换成合适的后端服务器的请求，窗口状态将按需要镜像到后端服务器。

DMX 系统显示图形的输出和窗口请求跨越多个屏幕，屏幕典型地被以网格方式安排，这样它们组成了一个大的 display。DMX 代码的输出部分由两部分组成，其中第 1 部分是前端代理 X 服务器（Xdmx），它接受客户端的连接，管理窗口并呈递原语，但实际上不显示任何绘画原语；第 2 部分是后端 X 服务器，它接收来自前端服务器的命令并在其屏幕上显示结果。

在 DMX 系统的输出初始化时，DMX 前端必须首先连接到每个后端 X 服务器，并且收集每个屏幕的信息。

旦这个屏幕初始化完成，每个后端 X 服务器的屏幕在整个屏幕中的位置被初始化。个全屏幕窗口在每个后端 X 服务器上被打开，并且每个屏幕上的光标被关闭。最终的 DMX 应用还能使用部分屏幕的窗口，或每个后端屏幕的多个窗口。

2．数据结构

屏幕资源用 xc/programs/Xserver/include/scrnintstr.h 中的结构 ScreenRec 描述，包括描述屏幕的各种参数变量及对屏幕的各种操作函数指针，该结构列出如下：

```
typedef struct _Screen {
    int         myNum;    /*在 Screens[]中这个实例的序号 */
    ATOM        id;
    short       width, height;
    short       mmWidth, mmHeight;
    short       numDepths;
    unsigned char         rootDepth;
    DepthPtr         allowedDepths;
    unsigned long         rootVisual;
    unsigned long    defColormap;
    short       minInstalledCmaps, maxInstalledCmaps;
    char             backingStoreSupport, saveUnderSupport;
    unsigned long    whitePixel, blackPixel;
    unsigned long    rgf; /* array of flags; she's -- HUNGARIAN */
    GCPtr       GCperDepth[MAXFORMATS+1];
      ......
    /* 随机的屏幕处理函数指针 */
    CloseScreenProcPtr       CloseScreen;
    QueryBestSizeProcPtr QueryBestSize;

    /*窗口处理函数指针*/
    CreateWindowProcPtr       CreateWindow;
    DestroyWindowProcPtr DestroyWindow;

    /*位图处理函数指针 */
    CreatePixmapProcPtr       CreatePixmap;
    DestroyPixmapProcPtr DestroyPixmap;
    /*回存处理函数指针 */
    SaveDoomedAreasProcPtr    SaveDoomedAreas;
    RestoreAreasProcPtr       RestoreAreas;

    /* 字体处理函数指针 */
    RealizeFontProcPtr        RealizeFont;
    UnrealizeFontProcPtr UnrealizeFont;
    /* 光标处理函数指针 */
    ConstrainCursorProcPtr ConstrainCursor;
    CursorLimitsProcPtr       CursorLimits;

    /* 图形环境处理函数指针 */
    CreateGCProcPtr      CreateGC;
    /*颜色映射处理函数指针 */
    CreateColormapProcPtrCreateColormap;
    DestroyColormapProcPtr    DestroyColormap;

    /*区域处理函数指针 */
#ifdef NEED_SCREEN_REGIONS
    RegionCreateProcPtr       RegionCreate;
    RegionInitProcPtr         RegionInit;
    RegionCopyProcPtr         RegionCopy;
    RegionDestroyProcPtr RegionDestroy;

#endif /* NEED_SCREEN_REGIONS */
    SendGraphicsExposeProcPtr SendGraphicsExpose;
    /* os 层处理函数指针 */
```

```
    ScreenBlockHandlerProcPtr BlockHandler;
    ScreenWakeupHandlerProcPtrWakeupHandler;
    ......
    ChangeBorderWidthProcPtr  ChangeBorderWidth;
    MarkUnrealizedWindowProcPtr    MarkUnrealizedWindow;
} ScreenRec;
```

X 支持同一个服务器连接多个屏幕，因此每个屏幕的信息放在一个数据结构 ScreenRec 中，每个屏幕对应的结构 ScreenRec 都有区域(region)、颜色映射(colormap)、光标(cursor)和字体(font)的处理例程入口。

结构 ScreenInfo 是一个全局的数据结构，包括指向所有屏幕的数组 screens[MAXSCREENS]。每个屏幕有个标识序号，它描述了所有屏幕显示的共同属性，如位图字节序等。结构 ScreenInfo（在 Xsever/include/scrnintstr.h 中）列出如下：

```
extern ScreenInfo screenInfo;
typedef struct _ScreenInfo {
    int         imageByteOrder;  // 图像字节次序
    int         bitmapScanlineUnit;  // 位图扫描线单位
    int         bitmapScanlinePad;
    int         bitmapBitOrder;  //位图 bit 次序
    int         numPixmapFormats;  // 位图格式数量
    PixmapFormatRec  formats[MAXFORMATS];  //位图格式
    int         arraySize;
    int         numScreens;  //屏幕数量
    ScreenPtr   screens[MAXSCREENS];
    int        numVideoScreens;
} ScreenInfo;
```

3. 输出初始化

输出初始化包括建立前端 X 服务器与后端 X 服务器的连接，初始化每个 X 服务器的所有屏幕和每个屏幕的数据结构。每个屏幕的数据结构组成数组后放入到全局变量 screeninfo 中，屏幕资源的 DMX 层表现为全局的结构数组 dmxScreens 。屏幕被初始化后，通过屏幕的数据结构 screenRec，DMX 层可访问每个屏幕的各种参数及例程。

函数 InitOutput 通过调用函数 AddScreen()初始化每个屏幕，下面列出函数 InitOutput 中与屏幕初始化的部分：

```
void InitOutput(ScreenInfo *pScreenInfo, int argc, char *argv[])
{
    int              i;
    static unsigned long dmxGeneration = 0;

    /* 调用函数 XOpenDisplay打开每个 display，建立与后端的X Server 之间的连接，并收集其信息.*/
    for (i = 0; i < dmxNumScreens; i++)
        dmxDisplayInit(&dmxScreens[i]);

    /* 因为到这里我们仅有一个屏幕，所以仅需要设置这个屏幕的位图格式*/
    if (!dmxSetPixmapFormats(pScreenInfo, &dmxScreens[0])) return;

    /* 想安装一个信号处理函数，允许在意想不到的信号后进行清除。 DIX/OS层已处理了 SIGINT和 SIGTERM信号，因此对于期望的信号来说，
所有的都是 OK。SIGHUP, SIGINT和 SIGTERM在 os/connection.c 中已被设置陷阱， SIGQUIT是另一个从键盘发送的通常信号。在这里设置
陷阱，以保证输入设备的键盘修改映射和其他状态被恢复（在某种程度上造成了不期望的 SIGQUIT 行为，因为它将与 SIGINT 行为相同，然而留下
输入设备的修改映射为空更是不期望的） */
    OsSignal(SIGQUIT, GiveUp);

    for (i = 0; i < dmxNumScreens; i++) {

        //使用例程 dmxScreenInit 初始化每个屏幕结构 ScreenRec，  并添加到 screenInfo 全局变量中
        AddScreen(dmxScreenInit, argc, argv);
```

```
    }

    /*计算每个屏幕的起始信息*/
    dmxInitOrigins();
    /*计算屏幕交迭信息 */
    dmxInitOverlap();
    /* 确认有一个全局可用的 width/height*/
    dmxComputeWidthHeight(DMX_NO_RECOMPUTE_BOUNDING_BOX);
    //字体初始化
    dmxInitFonts() ;
    ......
}
```

4. 屏幕数据结构 ScreenRec 中的区域（region）例程

一个区域是动态分配并用来描述一个存在于 XY 像素空间中不规则形状块的数据结构，可把它想像为屏幕上可被执行诸如 AND 和 OR 等操作的一套像素点。一个区域常被用做一个矩形的链表或包括选择的像素点的位图，样例服务器使用了 YX 带状的矩形作为区域。区域操作的例程指针位于 ScreenRec 结构中，区域的数据结构定义在 Xserver/include/regionstr.h 中,区域的操作代码在 Xserver/mi/miregion.c 中。

由于矩形的链表在尺寸上是没有限定的，所以区域的数据结构通常是一个大且动态分配的内存块。例如，在样例服务上，结构 RegionRec 有一个指向动态分配的矩形链表的指针。

区域的数据结构 RegionRec（在 Xserver/include/regionstr.h 中）列出如下：

```
typedef struct _Region RegionRec, *RegionPtr;
typedef struct _RegData {
   long   size;
   long   numRects;
} RegDataRec, *RegDataPtr;
//矩形区域坐标
typedef struct {
   short x1, x2, y1, y2;
} Box, BOX, BoxRec, *BoxPtr;
struct _Region {
   BoxRec      extents;    //坐标
   RegDataPtr  data;
};
```

大多数区域的操作有两种形式，即屏幕数据结构中的函数指针和宏定义的形式，程序在编译时选择其中一种操作方式。由于直接调用速度较快，所以大多数体系结构中使用了直接调用方式，举例如下：

```
RegionPtr pScreen->RegionCreate( rect, size)
BoxPtr rect;
int size;
```

宏定义的方式举例如下：

```
macro: RegionPtr REGION_CREATE(pScreen, rect, size)
```

屏幕数据结构中的创建区域的函数指针 RegionCreate 在屏幕初始化时就被赋上具体的例程，在这个服务器中赋的是函数 miRegionCreate。

函数 miRegionCreate（在 Xserver/mi/miregion.c 中）分配一个区域数据结构，并填充区域的矩形坐标 rect 及矩形数量 size，列出如下：

```
RegionPtr  miRegionCreate(rect, size)
           BoxPtr rect;
```

```
            int size;
{
    register RegionPtr pReg;

    pReg = (RegionPtr)xalloc(sizeof(RegionRec));  //分配区域数据结构的内存
    if (!pReg)
        return &miBrokenRegion;    //返回坐标为{0, 0, 0, 0}的区域
    if (rect)
    {
        pReg->extents = *rect;  //矩形区域
        pReg->data = (RegDataPtr)NULL;
    }
    else
    {
        pReg->extents = miEmptyBox; //坐标为{0, 0, 0, 0}的矩形
        if ((size > 1) && (pReg->data = xallocData(size)))  //分配内存
        {
            pReg->data->size = size;
            pReg->data->numRects = 0;
        }
        else
            pReg->data = &miEmptyData;
    }
    return(pReg);
}
```

屏幕数据结构 ScreenRec 中还有其他区域操作函数例程指针，它们都在屏幕初始化时被赋予具体的例程函数。如 RegionInit 例程初始化一个区域数据结构， RegionCopy 例程拷贝一个区域；RegionDestroy 销毁一个区域，Intersect 例程判断两个区域是否交叉，还有其他许多关于区域间运算的例程等。

5. 屏幕的光标例程

光标是绑定到指针设备的可见形式图形，默认光标是“ X”形状。光标可有任何形状。当一个客户端创建一个窗口时，它将根据屏幕上的窗口显示光标的形状。

数据结构 CursorRec 描述了光标的形状，它有一个指向 CursorBits 数据结构的指针，该数据结构存有光标图像的位图和一个在光标后面掩码的位图。 CursorRec 数据结构装有光标的前景色和背景色，为多个使用同一个字体以及描述 source 和 mask 的字形的 CursorRec 结构共享。

光标图像首先运用 mask，对背景颜色以 bit 位进行清除，然后覆盖写源图像的前景色。落在掩码图像 bit 上的源图像的bit 无定义，这样一个光标在前景和背景颜色中有透明和不透明部分。

X 允许任何光标尺寸，但一些硬件光标允许 M 像素点 N 像素点的最大化。因此可以将光标变换到一个较小的尺寸，但确认包括热点（hot-spot）

CursorBits结构在 Xserver/include/cursorstr.h 中是一个设备无关的结构，列出如下：

```
/*直接指到 bit位的 source和 mask，它们是服务器定义的位图格式*/
typedef struct _CursorBits {
    unsigned char *source;              /* points to bits */
    unsigned char *mask;           /* points to bits */
    Bool emptyMask;                /* 是否是全 0 掩码*/
    unsigned short width, height, xhot, yhot;     /* metrics */
    int refcnt;                    /* 共享用的引用计数 */
    pointer devPriv[MAXSCREENS];        /* 被 pScr->RealizeCursor设置*/
#ifdef ARGB_CURSOR
    CARD32 *argb;                      /*全颜色 alpha 混合*/
#endif
} CursorBits, *CursorBitsPtr;

typedef struct _Cursor {
```

```
    CursorBitsPtr bits;
    unsigned short foreRed, foreGreen, foreBlue; /*设备无关的前景颜色*/
    unsigned short backRed, backGreen, backBlue; /*设备无关的背景颜色*/
    int refcnt;
    pointer devPriv[MAXSCREENS];        /* 被pScr->RealizeCursor设置*/
#ifdef XFIXES
    CARD32 serialNumber;
    Atom name;
#endif
} CursorRec;
```

创建一个光标后，它为每个屏幕“realized”（重定义大小）。在实现时，每个屏幕有机会转换 bit 位到其他更方便的形式（如放置光标到离开屏幕的内存），并设置设备私有区域。在结构 CurorRec 或 CursorBits 中有设备私有区域指针数组 devPriv[MAXSCREENS]，例如，特殊屏幕和光标的私有项如下：

```
pCursor->devPriv[pScreen->myNum]
pCursor->bits->devPriv[pScreen->myNum]
```

在输出初始化时，屏幕数据结构会被赋上 dmxPointerSpriteFuncs（在 Xserver/hw/dmx/dmxcursor.c 中）的函数结构，这个结构列出如下：

```
miPointerSpriteFuncRec dmxPointerSpriteFuncs =
{
    dmxRealizeCursor,
    dmxUnrealizeCursor,
    dmxSetCursor,
    dmxMoveCursor,
};
```

函数 RealizeCursor （在 erver/hw/dmx/dmxcursor.c 中）在 DIX 需要光标时实现（分配和计算所有需要的数据）一个给定光标，函数 UnrealizeCursor 去掉一个给定的光标（释放动态分配的数据）。它们在与设备无关的光标被创建或销毁时被调用，调用方法如下：

```
pScreen->RealizeCursor( pScr, pCurs)
pScreen->UnrealizeCursor( pScr, pCurs)
```

函数 RealizeCursor 列出如下：

```
static Bool dmxRealizeCursor(ScreenPtr pScreen, CursorPtr pCursor)
{
    DMXScreenInfo *start = &dmxScreens[pScreen->myNum];
    DMXScreenInfo *pt;
    //分配光标私有结构dmxCursorPrivPtr的结构空间，并初始化
    if (!start->over || !dmxCursorDoMultiCursors || start->cursorNotShared)
        return _dmxRealizeCursor(pScreen, pCursor);

    for (pt = start->over; /* condition at end of loop */; pt = pt->over) {
        if (pt->cursorNotShared) continue;
        _dmxRealizeCursor(screenInfo.screens[pt->index], pCursor);
        if (pt == start) break;
    }
    return TRUE;
}
```

在屏幕的数据结构中还有其他光标操作例程，如 DisplayCursor 例程在屏幕上显示光标，RecolorCursor 例程改变当前显示光标的颜色，ConstrainCursor 例程限制光标在指定的区域中活动，SetCursorPosition 例程设置光标的位置等。

6. 屏幕的颜色映射

颜色映射是在像素值与显示在屏幕上的颜色的设备无关的映射，大多数客户端使用屏幕的默认颜色映射。一个屏幕的默认颜色映射在屏幕被初始化时初始化，它总保持存在而不属于任何常规客户端，并且常被客户端 0 （服务器自己）拥有。颜色映射中的项可被客户端应用程序修改。

开发者需要应用颜色映射例程处理设备相关的颜色映射，这些例程被放在 ScreenRec 数据结构的对应例程指针上。

下面以 CreateColormap 例程为例，在输出初始化时，ScreenRec 数据结构的 CreateColormap 例程指针上被赋予 fbInitializeColormap 函数来初始化颜色映射，ScreenRec 结构赋予 CreateColormap 例程的函数调用层次如图 1-12 所示。函数 fbSetupScreen 赋上了与帧缓冲操作的各种例程，包括窗口、位图及颜色映射等操作例程。

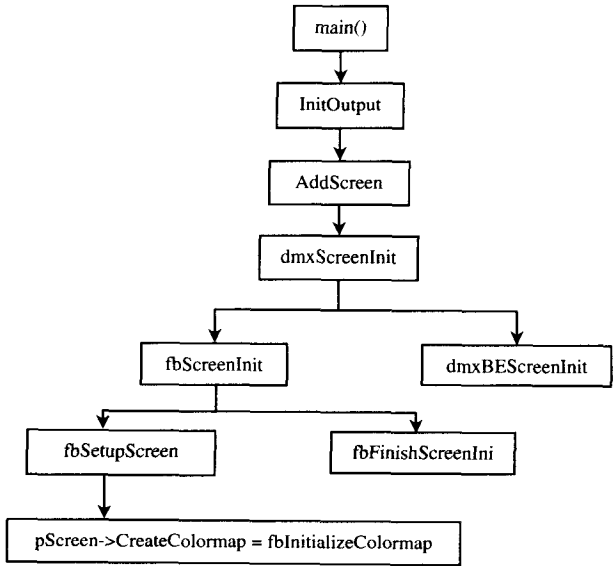


图 1-12 ScreenRec 结构赋予 CreateColormap 例程的函数调用层次

当一个客户端请求一个新的颜色映射和服务器创建默认颜色映射时， CreateColormap 例程被从 DIX 触发。这个例程为颜色映射分配内存和相关的存储，如客户端拥有的像素点的链表。然后调用 DDX 层的 CreateColormap 例程，在这个例程中初始化颜色映射。

函数 fbInitializeColormap（在 Xserver/fb/fbcmap.c 中）设置了像素值与显示在屏幕上的颜色的设备无关的映射方法，列出如下：

```
Bool fbInitializeColormap(ColormapPtr pmap)
{
   register unsigned i;
   register VisualPtr pVisual;
   unsigned lim, maxent, shift;

   pVisual = pmap->pVisual;
   lim = (1 << pVisual->bitsPerRGBValue) - 1;
   shift = 16 - pVisual->bitsPerRGBValue;
   maxent = pVisual->ColormapEntries - 1;
   if (pVisual->class == TrueColor) //真彩色
   {
        unsigned limr, limg, limb;
```

```
        limr = pVisual->redMask >> pVisual->offsetRed;
        limg = pVisual->greenMask >> pVisual->offsetGreen;
        limb = pVisual->blueMask >> pVisual->offsetBlue;
        for(i = 0; i <= maxent; i++)
        {
            /* rescale to [0..65535] then rgb bits */
           pmap->red[i].co.local.red =
                  ((((i * 65535) / limr) >> shift) * 65535) / lim;
           pmap->green[i].co.local.green =
                  ((((i * 65535) / limg) >> shift) * 65535) / lim;
           pmap->blue[i].co.local.blue =
                   ((((i * 65535) / limb) >> shift) * 65535) / lim;
        }
    }
    else if (pVisual->class == StaticColor)
    {
        unsigned    n;
        unsigned    r, g, b;
        unsigned    red, green, blue;

        for (n = 0; n*n*n < pVisual->ColormapEntries; n++);
        n--;
        i = 0;
        for (r = 0; r < n; r++)
        {
           red = (((r * 65535 / (n - 1)) >> shift) * 65535) / lim;
           for (g = 0; g < n; g++)
           {
                green = (((g * 65535 / (n - 1)) >> shift) * 65535) / lim;
                for (b = 0; b < n; b++)
                {
                blue = (((b * 65535 / (n - 1)) >> shift) * 65535) / lim;
                   pmap->red[i].co.local.red = red;
                   pmap->red[i].co.local.green = green;
                   pmap->red[i].co.local.blue = blue;
                   i++;
                }
           }
        }
        n = pVisual->ColormapEntries - i;
        for (r = 0; r < n; r++)
        {
           red = (((r * 65535 / (n - 1)) >> shift) * 65535) / lim;
           pmap->red[i].co.local.red = red;
           pmap->red[i].co.local.green = red;
           pmap->red[i].co.local.blue = red;
           i++;
        }
    }
    else if (pVisual->class == StaticGray)
    {
        for(i = 0; i <= maxent; i++)
        {
           /* rescale to [0..65535] then rgb bits */
           pmap->red[i].co.local.red = ((((i * 65535) / maxent) >> shift)
                        * 65535) / lim;
           pmap->red[i].co.local.green = pmap->red[i].co.local.red;
           pmap->red[i].co.local.blue = pmap->red[i].co.local.red;
        }
    }
    return TRUE;
}
```

ScreenRec 结构中与颜色映射有关的例程还有销毁颜色映射 DestoryColormap ，该例程的填充屏幕

上的颜色查找表的 InstallColormap 例程。通过这个查找表，颜色映射与参数 pColormap 中的颜色联系起来，以及改变颜色映射中的一些次值的 StoreColors 例程等。

7. 屏幕字体

字体是描述字符集中字符符号的位图，每个字符有一个位图。字体还有各种尺寸和参数，包括“bold”和“italic”。X 支持 8 位和 16 位字符编码的字体，Glyph （字形）是单个字符的位图。字体的位图格式是 BDF格式，是当前 Adobe 2.1 位图发布格式的轻微扩展。

所有的客户端从服务器得到字体信息，因此服务器应该能支持任何字体，字体可从服务器本地目录或从字体服务器得到。

8. 绘图

绘图包括被画的图形或显示在屏幕上的图形或内存中的位图，每个绘图结构 DrawableRec （在 Xserver/include/pixmapstr.h 中）有一个类型、类、与其相关的屏幕指针 ScreenPtr 、颜色深度、位置、尺寸和序列号等。类型有 DRAWABLE_PIXMAP，DRAWABLE_WINDOW 和 UNDRAWABLE_WINDOW，其中 UNDRAWABLE_WINDOW 仅用于窗口类输入。序列号用来决定绘图的惟一性，用来决定在 GC 中剪切信息的有效性。

绘图结构 DrawableRec 列出如下：

```
typedef struct _Drawable {
    unsigned char    type;      //绘制类型
    unsigned char    class;     //类型特定的类
    unsigned char    depth;     // 颜色深度
    unsigned char    bitsPerPixel;  //每个像素点的 bit 位数
    unsigned long    id;  //资源 id
    short       x;     //对于 window，屏幕绝对坐标 x；对于 pixmap，0
    short       y;     //对于 window 屏幕绝对坐标 y；对于 pixmap，0
    unsigned short   width; //宽度
    unsigned short   height;  //高度
    ScreenPtr        pScreen;  //屏幕的数据结构指针
    unsigned long    serialNumber;  //惟一的序列号
} DrawableRec;
```

PixmapRec（在 Xserver/include/pixmapstr.h 中）和 WindowRec 结构的第 1 个成员就是 DrawableRec。

位图是三元的图像 bit 位数组，依赖于设备。它有颜色深度指定每个像素点的 bit 位，所有的图像单元由像素组成。因此一个位图不能单独组成一个完整的图像，它仅代表一个矩形的像素值的数组。位图数据被按照其图形硬件建立后，可以被保存在任何位置，通常保存在服务器的动态分配空间中。

结构 PixmapRec 描述了位图的各种参数及位图 bit 位数据，列出如下：

```
typedef struct _Pixmap {
    DrawableRec     drawable;
    int         refcnt;
    /*位图位置的设备特定指示，在样例服务器中，所有的位图存在内存中，它保存位图以扫描线为单位的宽度*/
    int             devKind;
    DevUnion        devPrivate;  //  通常指向位图数据，但也能被 DDX 用做其他目的
#ifdef PIXPRIV
    /*数组中元素被 AllocatePixmapPrivateIndex()  分配，不同的图形硬件有不同的管理需要。如果图形总被与地址空间无关的处理器处理，将不需要指针指向 bit 图像本身*/
    DevUnion        *devPrivates; /* real devPrivates like gcs & windows */
#endif
#ifdef COMPOSITE
    short       screen_x;
    short       screen_y;
#endif
} PixmapRec;
```

在屏幕的数据结构中有位图的操作例程指针，它在输出初始化时被赋予具体的位图操作例程。这些例程有创建请求尺寸位图的 CreatePixmap 例程，它分配一个 PixmapRec 结构并填充这个结构的数据；销毁一个位图数据结构 DestroyPixmap 的例程，以及改位图的 PixmapRec 结构信息的 ModifyPixmapHeader 例程等。

窗口是屏幕上可见的或潜在可见的矩形， DIX 窗口函数维护一个内部树型数据结构，它代表当前的映射窗口的关系。

在树中的根窗口是在整个屏幕的的窗口，兄弟窗口组成了一个双向链表。窗口用结构 WindowRec（在 Xserver/include/windowstr.h 中）进行描述，该结构列出如下：

```
typedef struct _Window {
    DrawableRec     drawable;
    WindowPtr       parent;         // 父链表
    WindowPtr       nextSib;        //下一个兄弟
    WindowPtr       prevSib;        //上 个 兄 弟
    WindowPtr       firstChild;     //第 1 个孩子
    WindowPtr       lastChild;      //最后一个孩子
    RegionRec       clipList;       // 输出的剪切矩形
    RegionRec       borderClip;     /* NotClippedByChildren + border */
    union _Validate *valdata;
    RegionRec       winSize;        //窗口区域
    RegionRec       borderSize;     //边界区域
    DDXPointRec     origin;         // 相对于父亲的位置
    unsigned short  borderWidth;    // 边框宽度
    unsigned short  deliverableEvents;
    Mask        eventMask;
    PixUnion        background;
    PixUnion        border;
    pointer         backStorage;        // 当 BS取消时为 null
    WindowOptPtr    optional;
    unsigned        backgroundState:2; /* None, Relative, Pixel, Pixmap */
    unsigned        borderIsPixel:1;
    unsigned        cursorIsNone:1;     /* else real cursor (might inherit) */
    unsigned        backingStore:2;
    unsigned        saveUnder:1;
    unsigned        DIXsaveUnder:1;
    unsigned        bitGravity:4;
    unsigned        winGravity:4;
    unsigned        overrideRedirect:1;
    unsigned        visibility:2;
    unsigned        mapped:1;
    unsigned        realized:1;     /*  祖先所有的被映射 */
    unsigned        viewable:1;     /* realized && InputOutput */
    unsigned        dontPropagate:3;/* index into DontPropagateMasks */
    unsigned        forcedBS:1;     /*   系统支持的后备存储 */
#ifdef NEED_DBE_BUF_BITS
#define DBE_FRONT_BUFFER 1
#define DBE_BACK_BUFFER  0
    unsigned        dstBuffer:1;    /*绘制的目的缓存 */
    unsigned        srcBuffer:1;    /*绘制的源缓存 */
#endif
#ifdef COMPOSITE
    unsigned        redirectDraw:1;     /*绘制重定向*/
#endif
    DevUnion        *devPrivates;
} WindowRec;
```

窗口的边框由 DDX 例程绘制，窗口内容被客户端通过到服务器的请求绘制。窗口绘画是通过暴露事件系统进行，当一个区域被暴露时， DIX 产生一个暴露事件，通知客户端重画窗口并传递需要重画的最小区域。

就客户端来说，服务器可能保留窗口隐藏部分的输出在离开屏幕的内存中，称为“回存”。当这样窗口的部分变成暴露时，它能快速移像素点到触发一个暴露事件处进行代替，并等待客户端做出反应。拷贝窗口已存在的部分比重绘要快得多。

X提供了用于回存的 hook，回存的使用方法由服务器的应用决定，例如可以仅对用户请求的某个窗口使用回存或者使用回存直到内存用完。

DIX 层窗口函数维护了一个内部的窗口树型结构，它代表了当前映射窗口的关系。当窗口在屏幕显示时，必须通过映射窗口实现。当 DIX请求 DDX绘制窗口边框时，窗口的边框才被绘制。窗口的内容通过客户端发请求到服务器而由 DDX 层绘制。

当客户端请求一个窗口操作时，如窗口创建或移动，一个新的状态被计算。在转变期间， DIX 通知 DDX 窗口的哪个矩形将变成隐藏，哪个变成暴露，为此提供一个回存的 hook。如果 DDX 不能恢复暴露的区域，DIX 产生暴露事件给客户端。然后客户端负责绘制暴露的窗口部分，而不是恢复。

如果一个窗口被重设置大小，像素点可能需要被移动。窗户端能请求“Gravity”，以至于窗口的某一块被移动。例如，如果窗口有控件或其他项目总是挂在窗口的边缘上，并且这条边在窗口重设置大小时需要移动，那么这些像素点应该被移动以避免客户端重画它。

在绘画操作中用到如下窗口数据结构。

（1）ClipList：RegionRec 结构变量，是 gc 中客户端裁剪区域相关的区域，用来裁剪输出。 clipList 有从它提取的窗口的多个孩子，还有交迭在这个窗口上的兄弟窗口的碎片。例程 NotClippedByChildren (pWin)返回未裁剪的部分。

（2）borderClip：RegionRec 结构变量，是 CopyWindow 使用的区域，其中包括窗口的区域、窗口的孩子和边框。它是 NotClippedByChildren + 边框。

（3）clipList 和 borderClip：在窗口中的位置关系如图 1-13 所示。

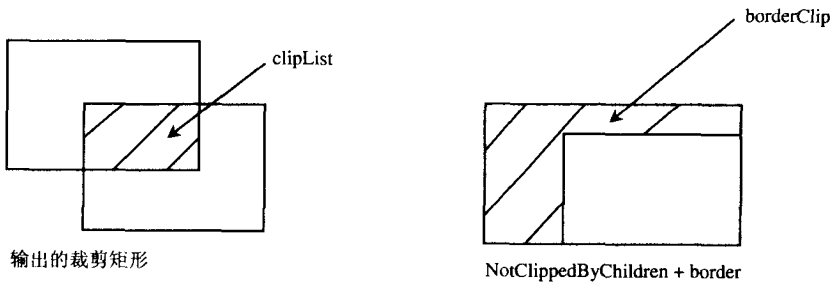


图 1-13 clipList 和 borderClip 在窗口中的位置关系

窗口结构 WindowRec 的其他大多数成员仅被 DIX 使用。

在 DIX 层中提供了各种窗口操作例程，它们是客户端对窗口操作的请求的服务例程。同时，在屏幕的数据结构中还提供了各种屏幕操作例程指针用来完成对屏幕的数据结构中的私有域进行操作。下面以 CreateWindow 为例进行说明。

例程 CreateWindow 在 DIX 层对客户端的创建窗口请求做出反应，它完成创建窗口的操作并且填充结构 WindowRec 中的各成员，许多属性来自父窗口。同时分发事件，最后返回创建的窗口。该例程列出如下：

```
DISPATCH_PROC(ProcCreateWindow);
int ProcCreateWindow(client)
   register ClientPtr client;
{
   register WindowPtr pParent, pWin;
   REQUEST(xCreateWindowReq);
   int result;
   int len;
```

```
    REQUEST_AT_LEAST_SIZE(xCreateWindowReq);

    LEGAL_NEW_RESOURCE(stuff->wid, client);
    if (!(pParent = (WindowPtr)SecurityLookupWindow(stuff->parent, client,
                                  SecurityWriteAccess)))
       return BadWindow;
    len = client->req_len - (sizeof(xCreateWindowReq) >> 2);
    if (Ones(stuff->mask) != len)
       return BadLength;
    if (!stuff->width || !stuff->height)
    {
      client->errorValue = 0;
       return BadValue;
    }
    pWin = CreateWindow(stuff->wid, pParent, stuff->x,
                    stuff->y, stuff->width, stuff->height,
                    stuff->borderWidth, stuff->class,
                    stuff->mask, (XID *) &stuff[1],
                    (int)stuff->depth,
                    client, stuff->visual, &result);
    if (pWin)
    {
         Mask mask = pWin->eventMask;

         pWin->eventMask = 0; /* subterfuge in case AddResource fails */
         if (!AddResource(stuff->wid, RT_WINDOW, (pointer)pWin))
            return BadAlloc;
         pWin->eventMask = mask;
    }
    if (client->noClientException != Success)
       return(client->noClientException);
    else
       return(result);
}
```

例程 CreateWindow （在 Xserver/dix/window.c 中）列出如下：

```
WindowPtr  CreateWindow(wid, pParent, x, y, w, h, bw, class, vmask, vlist,
          depth, client, visual, error)
    Window wid;                 //窗口 ID
    register WindowPtr pParent;      //父窗口的数据结构指针
    int x,y;                    //坐标
    unsigned int w, h, bw;      //   宽度和高度，边框宽度
    unsigned int class;            //类，为输入输出型、仅输入型或从父窗口继承
    register Mask vmask;
    XID *vlist;
    int depth;
    ClientPtr client;
    VisualID visual;
    int *error;
{
    ……
    pScreen = pParent->drawable.pScreen;
    if ((class == InputOutput) && (depth == 0))
      depth = pParent->drawable.depth;
    ancwopt = pParent->optional;
    if (!ancwopt)
          //查找给定窗口的祖先，查看是否装有 WindowOpt 结构。如果有，返回该窗口的指针
           ancwopt = FindWindowWithOptional(pParent)->optional;
    if (visual == CopyFromParent) {

         visual = ancwopt->visual;
    }
```

```
    /*查看 depth和 visual 是否为这个屏幕可接受的*/
    if ((visual != ancwopt->visual) || (depth != pParent->drawable.depth))
    {
         fOK = FALSE;
         for(idepth = 0; idepth < pScreen->numDepths; idepth++)
         {
            pDepth = (DepthPtr) &pScreen->allowedDepths[idepth];
            if ((depth == pDepth->depth) || (depth == 0))
            {
                 for (ivisual = 0; ivisual < pDepth->numVids; ivisual++)
                 {
                    if (visual == pDepth->vids[ivisual])
                    {
                          fOK = TRUE;
                          break;
                    }
                 }
            }
         }
         if (fOK == FALSE)
         {
            *error = BadMatch;
            return NullWindow;
         }
    }


    //分配窗口数据结构 WindowRec, 初始化 devPrivates 成员
    pWin = AllocateWindow(pScreen);

    pWin->drawable = pParent->drawable;
    pWin->drawable.depth = depth;
    if (depth == pParent->drawable.depth)
         pWin->drawable.bitsPerPixel = pParent->drawable.bitsPerPixel;
    else
{
         //查找颜色深度 depth 对应的位图格式
         for (format = screenInfo.formats; format->depth != depth; format++);
         pWin->drawable.bitsPerPixel = format->bitsPerPixel;
    }
    if (class == InputOnly)
              pWin->drawable.type = (short) UNDRAWABLE_WINDOW;
    pWin->drawable.serialNumber = NEXT_SERIAL_NUMBER;

    pWin->drawable.id = wid;  //  窗口 ID
    pWin->drawable.class = class;

    pWin->parent = pParent;
    SetWindowToDefaults(pWin);  //设置窗口数据结构中一些成员为默认值，通常为 null
…
    if (visual != ancwopt->visual)
{
        …
         pWin->optional->visual = visual;
         pWin->optional->colormap = None;
    }
    pWin->borderWidth = bw;
    …
    pWin->backgroundState = None;

    pWin->borderIsPixel = pParent->borderIsPixel;
    pWin->border = pParent->border;  //设置为父窗口的边框
    if (pWin->borderIsPixel == FALSE)
           pWin->border.pixmap->refcnt++;
```

```
    pWin->origin.x = x + (int)bw;
    pWin->origin.y = y + (int)bw;
    pWin->drawable.width = w;
    pWin->drawable.height = h;
    pWin->drawable.x = pParent->drawable.x + x + (int)bw;
    pWin->drawable.y = pParent->drawable.y + y + (int)bw;

      /* 为清楚的 WindowPtr建立正确的 clip list*/
    REGION_NULL(pScreen, &pWin->clipList);  //null  区域，即区域大小为 0
    REGION_NULL(pScreen, &pWin->borderClip);
    REGION_NULL(pScreen, &pWin->winSize);
    REGION_NULL(pScreen, &pWin->borderSize);
   /*将创建的窗口 pWin 插入链表*/
    pHead = RealChildHead(pParent);  //     返回第 1个孩子窗口或空窗口
    if (pHead)
    {
         pWin->nextSib = pHead->nextSib;
         if (pHead->nextSib)
            pHead->nextSib->prevSib = pWin;
         else
            pParent->lastChild = pWin;
         pHead->nextSib = pWin;
         pWin->prevSib = pHead;
    }
    else
    {
         pWin->nextSib = pParent->firstChild;
         if (pParent->firstChild)
            pParent->firstChild->prevSib = pWin;
         else
            pParent->lastChild = pWin;
         pParent->firstChild = pWin;
    }

    SetWinSize (pWin);
    SetBorderSize (pWin);

    if (!(*pScreen->CreateWindow)(pWin)) //  初始化窗口的私有域

    (*pScreen->PositionWindow)(pWin, pWin->drawable.x, pWin-    >drawable.y);

    if (!(vmask & CWEventMask))
         RecalculateDeliverableEvents(pWin); //重新判断事件的掩码

if (vmask)
/*改变窗口属性，值掩码 vmask 定义了哪个属性将被改变。值链表中含有的值，每个对应 vmask 中的一个 bit，从掩码中最不重要的到最重
要的*/
         *error = ChangeWindowAttributes(pWin, vmask, vlist, wClient (pWin));
    else
         *error = Success;
   ...
    if (!(vmask & CWBackingStore) && (defaultBackingStore != NotUseful))
    {
         XID value = defaultBackingStore;
         (void)ChangeWindowAttributes(pWin, CWBackingStore, &value, wClient (pWin));
         pWin->forcedBS = TRUE;
    }

    if (SubSend(pParent))  //  进行事件掩码比较，是其他事件掩码且是 SubstructureNotifyMask
    {//设置并分发事件
         event.u.u.type = CreateNotify;
         event.u.createNotify.window = wid;
         event.u.createNotify.parent = pParent->drawable.id;
         event.u.createNotify.x = x;
```

```
        event.u.createNotify.y = y;
        event.u.createNotify.width = w;
        event.u.createNotify.height = h;
        event.u.createNotify.borderWidth = bw;
        event.u.createNotify.override = pWin->overrideRedirect;
        DeliverEvents(pParent, &event, 1, NullWindow);
    }
    return pWin;
}
```

服务器设备独立部分拥有窗口树，但硬件可能想知道映射的窗口的关系，在 ScreenRec 结构中有指向例程的指针为硬件更新其内部状态提供一个机会。这些仅是 DDX 的辅助函数，它们不改变窗口树，窗口树仅被 DIX 层改变。

pScreen->CreateWindow(pWin) 例程是 DIX 创建一个窗口的 hook，它填充窗口的私有域 devPrivate。

当选择 cfbSetupScreen 函数初始化屏幕时，它被赋予操作窗口私有域的 CreateWindow 例程，cfbSetupScreen 函数中这部分操作代码（在 Xserver/cfb\cfbscrinit.c 中）列出如下：

```
Bool  cfbSetupScreen(pScreen, pbits, xsize, ysize, dpix, dpiy, width)
    register ScreenPtr pScreen;
    pointer pbits;        /*屏幕位图指针*/
    int xsize, ysize;     /*  像素点数  */
    int dpix, dpiy;       /*  每英寸像素点数  */
    int width;            /*  帧缓冲中的像素点宽度*/
{

     pScreen->CreateWindow = cfbCreateWindow;

}
```

初始化窗口的私有域数据结构 cfbPrivWin 例程 cfbCreateWindow（在 Xserver/cfb/cfbwindow.c 中）列出如下：

```
Bool  cfbCreateWindow(pWin)
    WindowPtr pWin;
{
    cfbPrivWin *pPrivWin;

    pPrivWin = cfbGetWindowPrivate(pWin); //得到窗口的私有结构 cfbPrivWin
    //填充窗口的私有数据结构
    pPrivWin->pRotatedBorder = NullPixmap;
    pPrivWin->pRotatedBackground = NullPixmap;
    pPrivWin->fastBackground = FALSE;
    pPrivWin->fastBorder = FALSE;
    pPrivWin->oldRotate.x = 0;
    pPrivWin->oldRotate.y = 0;

#ifdef PIXMAP_PER_WINDOW
    /*建立到屏幕位图的指针*/
    pWin->devPrivates[frameWindowPrivateIndex].ptr =
      (pointer) cfbGetScreenPixmap(pWin->drawable.pScreen);
#endif
    return TRUE;
}
```

屏幕数据结构还提供了其他对窗口私有域进行操作的例程，如 DestoryWindow 例程销毁窗口私有数据结构；PositionWindow 例程当窗口位置改变时，执行一些窗口私有域中的特殊操作等。

屏幕数据结构还提供了窗口绘画例程直接处理实际的窗口图像，如 ClearToBackground 例程、PaintWindowBackground 例程、PaintWindowBorder 例程和 CoypWindow 例程等。

屏幕数据结构还提供了回存例程接口，如 SaveAreas 例程回存新隐藏的区域到回存指定内存中，RestoreAreas例程从回存内存中恢复保存的区域， ClearBackingStore 例程清除回存等。

屏幕数据结构还提供了多层帧缓冲屏幕操作例程接口，如果用户有一个带有多套独立的 bit 位层平面，如上 层、下层和主层平面，多层帧缓冲屏幕操作例程将被用到，如 MarkWindow 例程及 ChangeSaveUnder 等。

9．图形上下文（graphic context，GC）

图形上下文结构装有状态参数，如前景和背景的像素值（颜色）、当前线条的样式和宽度、产生模式的当前 tile 或 stipple，以及产生文本的当前字体及其他类似属性。 GC 结构还装有改变 GC 状态的函数指针结构和 GC 的基本图形操作函数指针结构。

drawable（绘图）与 GC 的区别是其描述了一个书写表面，并且书写可能已在上面进行；而 GC 描述了绘画进程。形象地说， drawable 如同是一块粉笔板，而 GC 如同一根粉笔。

GC 结构部分（在 Xserver/include/gcstruct.h 中）列出如下：

```
typedef struct _GC {
    ScreenPtr        pScreen;
    unsigned char    depth;
    unsigned char    alu;
    unsigned short   lineWidth;
    unsigned short   dashOffset;
    ......
    unsigned long    fgPixel;
    unsigned long    bgPixel;

    PixUnion         tile;
    PixmapPtr        stipple;
    DDXPointRec      patOrg;            /* (tile, stipple)  的起点 */
    struct _Font     *font;             //      字体
    DDXPointRec      clipOrg;
    DDXPointRec      lastWinOrg;        /*  上次有效窗口的位置*/
    pointer          clientClip;
    unsigned long    stateChanges;  /* 与 GC_<kind>进行掩码 */
    unsigned long      serialNumber;
    //修改 GC 的状态函数结构，包含 ChangeGC、CopyGC 及 ChangeClip 等例程指针
    GCFuncs          *funcs;
    /*GC 触发的执行基本图形操作函数指针结构，包含 FillSpans、CopyArea、Polylines、PolyArc 及 FillPolygon 等例程*/
    GCOps        *ops;
    DevUnion         *devPrivates;

    PixmapPtr        pRotatedPixmap;  /*为对齐而执行 tile/stipple 旋转*/
    RegionPtr        pCompositeClip;
    /* fExpose & freeCompClip defined above */
} GC;
```

下面以例程 CreateGC 为例，说明图形上下文的例程传递过程。

当客户端要求建立图形上下文的请求传递到服务器时，服务器将请求分发到例程 ProcCreateGC。例程 ProcCreateGC 对请求保护判断后，调用了 DIX 层的 CreateGC 例程。

DIX 层的 CreateGC 例程完成与具体设备无关的操作，分配 GC 结构空间并设置与设备无关的一些参数。然后调用 (*pGC->pScreen->CreateGC)(pGC)初始化成员 GCFuncs 和 GCOps，并赋予与设备相关的图形绘制函数和图形上下文属性改变函数。

例程 ProcCreateGC（在 Xserver/dix/dispatch.c 中）列出如下：

```
DISPATCH_PROC(ProcCreateGC);
ProcCreateGC(client)
    register ClientPtr client;
```

```
{
    int error;
    GC *pGC;
    register DrawablePtr pDraw;
    unsigned len;
    //即 register xCreateGCReq  *stuff = (xCreateGCReq *)client->requestBuffer
    REQUEST(xCreateGCReq);

    REQUEST_AT_LEAST_SIZE(xCreateGCReq); //    判断请求 xCreateGCReq 的长度是否合适
    client->errorValue = stuff->gc;
    LEGAL_NEW_RESOURCE(stuff->gc, client);  //  判断 stuff->gc 是否有正确 ID
    //验证是否是可绘制的窗口
    SECURITY_VERIFY_DRAWABLE (pDraw, stuff->drawable, client,
                  SecurityReadAccess);
    len = client->req_len -  (sizeof(xCreateGCReq) >> 2);
    if (len != Ones(stuff->mask))
       return BadLength;

    //创建环境上下文
    pGC = (GC *)CreateGC(pDraw, stuff->mask,       (XID *) &stuff[1], &error);
    if (error != Success)
       return error;
    if (!AddResource(stuff->gc, RT_GC, (pointer)pGC))
          return (BadAlloc);
    return(client->noClientException);
}

GCPtr  CreateGC(pDrawable, mask, pval, pStatus)
    DrawablePtr pDrawable;
    BITS32 mask;
    XID         *pval;
    int         *pStatus;
{
    register GCPtr pGC;

   pGC = AllocateGC(pDrawable->pScreen); //分配 GC结构空间,填充 pGC->devPrivates

    //初始化 pGC
    pGC->pScreen = pDrawable->pScreen;
    pGC->depth = pDrawable- >depth;

    if (mask & GCForeground)  //前景
    {
          pGC->tileIsPixel = FALSE;
        pGC->tile.pixmap = NullPixmap;
    }
    else
     {
         pGC->tileIsPixel = TRUE;
         pGC->tile.pixel = 0;
    }

   pGC->patOrg.x = 0;
   pGC->patOrg.y = 0;

   /*使用默认的 font和 stipple */
    pGC->font = defaultFont;
    defaultFont->refcnt++;
    pGC->stipple = pGC->pScreen->PixmapPerDepth[0];
    pGC->stipple->refcnt++;

   pGC->stateChanges = (1 << (GCLastBit+1)) - 1;
   if (!(*pGC->pScreen->CreateGC)(pGC))  //初始化成员 GCFuncs  和 GCOps
        *pStatus = BadAlloc;
   else if (mask)
```

```
        *pStatus = ChangeGC(pGC, mask, pval);
    else
        *pStatus = Success;
  ......
    return (pGC);
}
```

当选择 cfbSetupScreen 函数初始化屏幕时，它被赋予操作窗口私有域的 cfbCreateGC 例程，cfbSetupScreen 函数中这部分操作代码（在 Xserver/cfb\cfbscrinit.c 中）列出如下：

```
Bool  cfbSetupScreen(pScreen, pbits, xsize, ysize, dpix, dpiy, width)
    register ScreenPtr pScreen;
    pointer pbits;          /*屏幕位图指针*/
    int xsize, ysize;      /*  像素点数 */
    int dpix, dpiy;         /* 每英寸像素点数 */
    int width;              /* 帧缓冲中的像素点宽度*/
{
    ......
    pScreen->CreateWindow = cfbCreateGC; ;
    ......
}
```

函数 cfbCreateGC 为图形上下文赋予基本图形绘制函数和改变图形上下文属性的函数，并初始化图形上下文的私有域，函数 cfbCreateGC（在 Xserver/cfb/cfbgc.c 中）列出如下：

```
Bool  cfbCreateGC(pGC)
    register GCPtr pGC;
{
    cfbPrivGC  *pPriv;

    if (PixmapWidthPaddingInfo[pGC->depth].padPixelsLog2 == LOG2_BITMAP_PAD)
        return (mfbCreateGC(pGC));
    pGC->clientClip = NULL;
    pGC->clientClipType = CT_NONE;

    /*一些输出原语不需要，因为 dix/CreateGC()设置了所有的变化 bit，它们被填充在 ValidateGC 中。其他是必需的，因为虽然它们依赖颜色帧缓冲，但其没有变化*/
    pGC->ops = &cfbNonTEOps;  // 图形上下文基本绘制操作函数
    pGC->funcs = &cfbGCFuncs;  //改变图形上下文属性的函数

    /* cfb想在扫描转换之前转换 */
    pGC->miTranslate = 1;

    pPriv = cfbGetGCPrivate(pGC);
    pPriv->rop = pGC->alu;
    pPriv->oneRect = FALSE;
    pGC->fExpose = TRUE;
    pGC->freeCompClip = FALSE;
    pGC->pRotatedPixmap = (PixmapPtr) NULL;
    return TRUE;
}
```

X 协议使用原语操作完成图形绘制，称为“绘制原语”。绘制原语在 Xlib 中表现为绘制函数，包括线条绘制、区域填充、弧线和文本绘制等。客户端给出这些绘制请求，在服务器端必须实现这些例程，这些例程在服务器端被赋给图形上下文中的 pGC->ops。

根据服务器的不同可以不同，但服务器必须提供基本图形绘制例程的实现。在样例服务器中，将基本图形绘制例程进行分层。与设备无关的绘制部分代码放在 Xserver/mi 目录下，mi 是 machine independ 的缩写。基本图形绘制例程的上层绘制例程接口及下层直接操作像素的例程接口，在 pGC->ops 结构中都有定义。

与设备有关的绘制部分是像素点的绘制方法，称为“Pixblit”。Pixblit 例程读写像素点的值，并且直接与图像数据打交道。这个服务器中的 Pixblit 例程放在 Xserver/mfb（在单色帧缓冲中，mfb 是 Monochrome Frame Buffer缩写）和 Xserver/cfb（在彩色帧缓冲中，cfb 是 Color Frame Buffer 缩写）。如果使用一个正常的帧缓冲类型显示，则可以使用 mfb或 cfg 和 mi 中的代码完成基本图形绘制：否则用户必须提供 Pixblit 例程。

Pixblit 例程被分成 Span 例程、Glypth 例程和 PushPixels 例程 3 个子集，Span 例程简单地填充像素行；Glypth 例程填充字符的图形；PushPixels 例程是为更多复杂的图像创建提供的 bitblt 位操作函数。Glypth 例程和 PushPixels 例程实际上有一些设备无关的应用，它们依赖于 Span 例程。

这些例程中所有的图形操作最终变为对“ spans ”的基本操作。一个 span 是像素点的水平方向的一段。图形上下文提供了对 spans 的操作例程指针，这些例程列出如下：

```
void pGC->ops->FillSpans(dst, pGC, nSpans, pPoints, pWidths, sorted)
void pGC->ops->SetSpans(pDrawable, pGC, pSrc, ppt, pWidths, nSpans, sorted)
void pScreen->GetSpans(pDrawable, wMax, pPoints, pWidths, nSpans)
```

Glyph 例程为客户端的文字绘制请求绘制单个的字符，图形上下文提供了对 Glyph 的操作例程指针，这些例程列出如下：

```
void pGC->ops->PolyGlyphBlt(pDrawable, pGC, x, y, nglyph, ppci, pglyphBase)
void pGC->ops->ImageGlyphBlt(pDrawable, pGC, x, y, nglyph, ppci, pglyphBase)
```

这些例程必须在 pglyphBase 中定义的位图字形和在 ppci 中定义的字体点阵到 DrawbalePtr 和 pDrawable 中。

PushPixels 例程以位图定义的某种形状写当前的填充样式到 drawable 中，它等价于使用第 2 次 stipple 通过一个模板推行 fillStyle，例如，支持的模板是 00111100，stipple 是 10101010，PushPixels 结果是 00101000

图形上下文提供了对 PushPixels 的操作例程指针，列出如下：

```
void pGC->ops->PushPixels(pGC, pBitMap, pDrawable, dx, dy, xOrg, yOrg)
```

第 2 章　Qt 编程核心技术

Qt/X11 与 Qt/Embedded 版本的 Qt 类库尽管类的底层实现有一些细微区别，但类的声明及方法函数相同。对于编写 Qt 应用程序的开发者来说，几乎不需要关心这些区别，使用的类与方法函数接口是一样的。本章介绍了 Qt 对象模型、国际化方法、元对象及代码生成、进程间通信、窗口部件的基类、模板库和集合类、Qt 线程、鼠标拖放、键盘焦点、会话管理，以及调试等方面技术。

2.1　Qt 概述

Qt 是一个跨平台的 C++图形用户界面库，由挪威 TrollTech 公司出品，目前包括 Qt/X11、基于 Framebuffer 的 Qt Embedded、快速开发工具 Qt Designer 及国际化工具 Qt Linguist 等。Qt 支持 Unix 及 Linux 系统，还支持 Windows NT/ Windows 2k 及 Windows 95/98 平台。Qt 的良好封装机制使其模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。 Qt API 和开发工具对所有支持平台都是一致的，从而可以进行独立于平台的程序开发和配置。它使得跨平台软件编程直观、简易和方便。Qt 提供了一种称为“ signals/slots ”的安全类型来替代 callback 回调函数，使得各个控件之间的协同工作变得十分简单。

Qt 在 Linux 下有 GPL 版，可方便用户的学习及开发。如果用户使用 C++，对库的稳定性及健壮性要求比较高，并且希望跨平台开发，那么使用 Qt 是较好的选择。Qt 还支持 2D/3D 图形渲染、OpenGL 及 XML 等。

Qt Script for Applications（QSA） 是 Trolltech 的跨平台脚本工具箱，Qt 为静态的 Qt/C++程序提供了一个脚本界面，可以定制和扩展程序。

Qtopia 是为基于 Linux 的 PDA、智能电话和其他移动设备设计的一个全面并且可以用户化的应用程序平台和用户界面。

Qt/Embedded 是面向嵌入式系统的 Qt 版本，是 Qt 的嵌入式 Linux 窗口，是完整的自包含 C++ GUI 和基于 Linux 的嵌入式平台开发工具。Qt/Embedded API 可用于多种开发项目。许多基于 Qt 的 X Window 程序可以非常方便地移植到嵌入式版本，适用于高端 PDA 等产品。Qt/Embedded 内部对于字符集的处理采用了 UNICODE 编码标准。

Qt 是基于面向对象的 C++语言，它提供了 signal（信号）和 slot （槽）的对象通信机制，具有可查询和可设计的属性以及强大的事件和事件过滤器。同时，它还具有字符国际化，即支持根据上下文进行国际化字符串翻译。许多 Qt 特性基于 QObject 的继承，通过标准 C++技术实现。

2.2　Qt 对象模型

标准的 C++对象模型提供了非常有效的对运行时参数的支持，但 C++对象模型的静态特性在某些问题上缺乏灵活性。图形用户界面编程需要运行的高效和高层次的灵活性， Qt 提供了 C++的高速及 Qt 对象模型的灵活性，Qt 增加了如下特性。

（1）非常有效的对象通信，即 signal 与 slot。

（2）可查询和可设计的对象属性。

（3）有效的事件及事件过滤器。

（4）为国际化提供了上下文式的字符串翻译。

（5）定时器使得在一个事件驱动的 GUI 中整合多个任务成为可能。

（6）层次化并可查询的对象树以对象继承关系这样自然的方式组织对象。

（7）保护指针 QGuardedPtr 在引用对象销毁时自动设置为 0，而正常的 C++指针在其对象销毁时变成危险的指针。

Qt 的这些特征基于 QObject 的继承性应用在标准 C++技术上。另外，为对象通信机制和动态属性系统需要 Qt 自己的元对象编译器 moc（Meta Object Compiler）提供了元对象系统。

元对象系统是一个 C++扩展，这个扩展使得 Qt更适合于真正的组件 GUI编程。

组成 Qt 对象模型的基类说明如表 2-1 所示。

表 2-1 Qt 对象模型的基类说明

基 类	说 明
QGuardedPtr	一个模板类，提供了对 QObject 对象指针的保护
QMetaObject	关于 Qt 对象的元信息
QMetaProperty	存储有关属性的元数据
QObject	所有 Qt 对象的基类
QObjectCleanupHandler	监视多个 QObject 的生命周期
QVariant	用于大多数通用 Qt 数据类型的联合

其中 QGuardedPtr 类和 QObjectCleanupHandler 的使用方法说明如下。

一个保护的指针 QGuardedPtr<X>，除了在引用对象被销毁时能自动被设置为 0 外，其他在使用时如同一个正常的 C++指针 X*。而正常的 C++指针在对象销毁时变成了不确定的指针，其中 X 必须是 QObject 的一个子类。保护的指针在需要保存一个他人拥有的 QObject 指针时很有用，这个指针所指对象在还在持有其引用时，他人可能删除它，开发者可以安全地测试这个指针的有效性。

使用保护指针 QGuardedPtr<X>的方法见下例：

```
  QGuardedPtr<QLabel> label = new QLabel( 0, "label" );
label->setText( "I like guarded pointers" );

delete (QLabel*) label; // 模拟别人销毁了 label

if ( label)
  label->show();
else
  qDebug("The label has been destroyed");
```

程序将输出“The label has been destroyed”，而不是引用了一个无效的地址。

如果需要知道他人拥有的多个 QObject 何时已被删除，QObjectCleanupHandler 将是很有用的，如分配在一个共享库中的应用程序卸载共享库时必须知道共享库的所有对象必须销毁，一个使用 QObjectCleanupHandler来监视对象销毁的例子如下：

```
class FactoryComponent : public FactoryInterface, public QLibraryInterface
{
public:
  ...

  QObject *createObject();
```

```
    bool init();
    void cleanup();
    bool canUnload() const;

private:
    QObjectCleanupHandler objects;
};

// 分配一个新的对象并把它加入到cleanup handler中
QObject *FactoryComponent::createObject()
{
   return objects.add( new QObject() );
}

// QLibraryInterface接口应用
bool FactoryComponent::init()
{
   return TRUE;
}

void FactoryComponent::cleanup()
{
}

//当所有QObject对象被销毁时，卸载库才是安全的。
bool FactoryComponent::canUnload() const
{
    return objects.isEmpty();
}
```

元对象系统不能对信号与槽使用模板，一个简单的理由是由于各个编译器的不充分，所以 Qt 不能在多个平台的应用程序中完全应用模板。即使在今天，许多使用很广的 C++编译器应用到先进模板时仍有问题，例如，不能安全地依赖于部分模板实例。

即使 C++编译器对模板有优秀的支持，我们也不能抛弃元对象编译器使用的基于字符串访问。

这是因为 Qt 信号与槽的语法直观且易用易读，在类定义中声明信号，确保了信号在保护成员函数中被保护。

Qt 的元编译器 moc（Meta Object Compiler）产生能被任何标准 C++编译器访问的附加 C++代码。moc 读取 C++代码文件，如果它发现类声明中含有“ Q_OBJECT ”宏，则将为这些类产生另外的 C++代码，其中装有元对象代码。这些被 moc 产生有 C++源代码必须被编译连接到这个类（或被#include 包含进这个类的源文件中）。moc 通常不被手动调用，而是因编译系统自动调用，这样它不需要程序员做另外的工作。

由于为信号与槽增加了 moc，我们能在 Qt 中增加其他有用的内容，这是模板类不能做的。如使用 tr()函数翻译先进的属性系统和扩展的运行类型信息，属性系统可用于诸如 Qt Designer 这样的通用用户界面设计工具。带有 moc 预处理器的 C++基本上提供了面向对象的 C 的灵活性或类似 Java 的运行环境，并保持了 C++的执行效率和扩展性。

2.2.1 信号和槽

信号和槽用于对象间的通信，该机制是 Qt 的一个重要特征。在图形用户界面编程中，经常需要将一个窗口部件的变化通知给另一个窗口部件，或者说希望对象间进行通信。一般的图形用户界面编程中采用回调函数进行对象间通信，这样回调和处理函数捆绑在一起，没有信号和槽机制的简便和灵活。信号和槽的连接原理如图 2-1 所示。

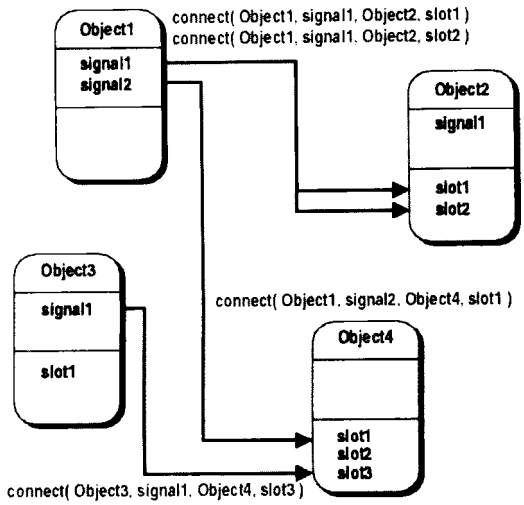


图 2-1　信号和槽的连接原理

Qt 的窗口部件有多个预定义的信号，槽是一个可以被调用处理特定信号的函数。 Qt 的窗口部件有多个预定义的槽，当一个特定事件发生的时候，一个信号被发射，对信号感兴趣的槽就会调用对应响应函数。

信号/槽机制在 QObject 类中实现，从 QObject 类或其一个子类（如 QWidget 类）继承的所有类可以包含信号和槽。当对象改变其状态的时候，信号被发送，对象不关心有没有其他对象接收到这信号。槽是类的正常成员函数，可以将信号和槽通过 connect 函数任意相连。当一个信号被发射，它所连接的槽会被立即执行，如同一个普通函数调用一样。信号与槽的连接示意如图 2-2 所示。

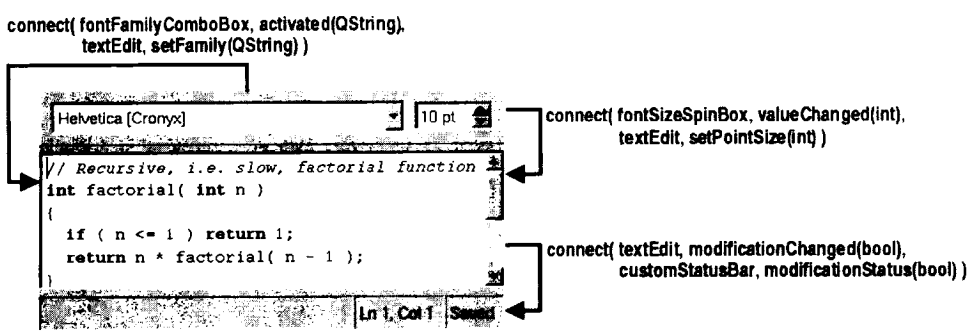


图 2-2　信号和槽的连接示意

一个带有信号和槽的 Qt 类 Foo 声明如下：

```
class Foo : public QObject
{
    Q_OBJECT  //包含信号和/或者槽的类必须声明 Q_OBJECT
 public:
    Foo();
    int value() const { return val; }
public slots:
    void setValue( int );  //槽的声明
 signals:
    void valueChanged( int );  // 信号声明
 private:
    int val;
```

```
};

void Foo::setValue( int v )
{
       if ( v != val ) {
          val = v;
          emit valueChanged(v); //发送信号
       }
}
```

下面把两个对象连接在一起：

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));//信号与槽连接
b.setValue( 11 ); // a == undefined  b == 11
a.setValue( 79 ); // a == 79        b == 79
b.value();
```

调用 a.setValue(79)会使 a 发射一个 valueChanged() 信号，b 将会在其 setValue()槽中接收这个信号，即 b.setValue(79)被调用。然后 b 会发射同样的 valueChanged() 信号，但是因为没有槽被连接到 b 的这个信号，所以没有发生任何事件（信号消失）。

signals、slots 和 emit 不是 C++的标准关键字，需要预处理程序改变或者移去这些关键字，然后使用标准的 C++编译器。对包含信号和槽的类定义运行 moc，生成一个可以和其他对象文件编译和连接成引用程序的 C++源文件。

槽是普通成员函数，它和普通成员函数一样分为 public、protected 和 private 共 3 类。public slots 表示声明的是任何信号都可以相连的槽； protected slots 表示这个类的槽及其子类的信号才能连接； private slots 表示这个类本身的信号可以连接这个类的槽。

元对象编译器解析一个 C++文件中的类声明并且生成初始化元对象的 C++代码，元对象包括所有信号和槽函数的名称，以及这些函数的指针。

2.2.2 元对象系统

Qt 中的元对象系统是用来处理对象间通信的信号/槽机制、运行时的类型信息和动态属性系统，它基于 QObject 类、类声明中的私有段中的 Q_OBJECT 宏和元对象编译器。

moc 读取 C++源文件，如果发现类的声明中含有 Q_OBJECT宏，则为含有 Q_OBJECT宏的类生成另一个含有元对象代码的 C++源文件，这个生成的源文件可以被类的源文件包含（#include）并和这个类的实现一起编译连接。

除了提供对象间通信的信号和槽机制（这是使用元对象最主要的原因）之外， QObject 中的元对象代码还实现其他特征：

className()函数在运行的时候以字符串返回类的名称，不需要 C++编译器中的本地运行类型信息的支持。

inherits()函数返回本对象一个在 QObject 继承树中一个特定类的实例。

tr()和 trUtf8()两个函数用于国际化中的字符串翻译。

setProperty() 和 property()函数用来通过名称动态设置并获得对象属性。

metaObject()函数返回这个类所关联的元对象。

在类的定义中声明了 Q_OBJECT宏，这个类才能使用元对象系统相关的特性。建议 QObject 的所有子类使用 Q_OBJECT宏，而不管它们是否实际使用了信号、槽和属性。

元对象编译器读取一个 C++源文件，如果发现其中的一个或多个类的声明中含有 Q_OBJECT 宏，则为这个使用 Q_OBJECT 宏的类生成另外一个包含元对象代码的 C++源文件。

如果使用 qmake 来生成 Makefile 文件，当需要的时候，编译规则中会包含调用元对象编译器，所以不需要直接使用元对象编译器。

元对象编译器生成的输出文件必须被编译和连接，如同用户程序中的其他的 C++代码一样，这种操作是用下述两种方式之一解决的。

方法 1：类的声明放在一个头文件（.h 文件）中。

如果在文件 myclass.h 中发现类的声明，元对象编译器的输出文件将会被放在一个称为“moc_myclass.cpp”的文件中。这个文件如普通文件一样被编译，输出对象文件的结果是 moc_myclass.o（在 Unix 下）或者 moc_myclass.obj（在 Windows 下）。然后这个对象会被包含到一个对象文件列表中，它们将在程序的最后连接阶段被连接在一起。

方法 2：类的声明放在一个实现文件（.cpp 文件）中。

如果在文件 myclass.cpp 中发现类的声明，元对象编译器的输出文件将会被放在一个称为“myclass.moc”的文件中。这个文件需要被实现文件包含（#include），即 myclass.cpp 需要包含下面这行并放在所有的代码之后：

```
#include "myclass.moc"
```

这样，元对象编译器生成的代码将会和 myclass.cpp 中普通的类定义一起被编译和连接。

方法 1 是常规的方法；方法 2 用在实现文件自包含，或者 Q_OBJECT 类是内部实现并且在头文件中不可见的这些情况下。

建议使用自由 makefile 生成工具 qmake 来生成自己的 Makefile，这个工具可以识别方法 1 和方法 2 风格的源文件，并建立一个可以执行所有必要的元对象编译操作的 Makefile。

如果需要建立自己的 Makefile，根据上面的方法 1 和方法 2 所提的声明 Q_OBJECT 宏不同方法，下面说明如何在 makefile 中包含元对象编译操作。

对于在头文件中声明了 Q_OBJECT 宏的类，如果只使用 GNU 的 make，则在 makefile 中如下添加元对象编译规则：

```
moc_%.cpp: %.h
	moc $< -o $@
```

还可以按下面的格式编写单个文件的规则：

```
moc_NAME.cpp: NAME.h
	moc $< -o $@
```

开发者必须把 moc_NAME.cpp 添加到自己的 SOURCES 变量中并且把 moc_NAME.o（Linux 操作系统）或者 moc_NAME.obj（Windows 操作系统）添加到 OBJECTS 变量中。

对于在实现文件（.cpp 文件）中声明 Q_OBJECT 的类，则可使用下面这样的 makefile 规则：

```
NAME.o: NAME.moc

NAME.moc: NAME.cpp
	moc -i $< -o $@
```

这样将会保证 make 程序会在编译 NAME.cpp 之前运行元对象编译器，然后可以把下面这行放在 NAME.cpp 的末尾，这样在这个文件中的所有的类声明都知道这个元对象：

```
#include "NAME.moc"
```

元对象编译中常出现的错误是：

```
YourClass::className() is undefined
```

或者：

```
YourClass lacks a vtbl
```

出现这种错误的绝大多数情况是忘记了编译或者 #include 元对象编译器产生的 C++代码，或者（在前面的情况下）没有在连接命令中包含相应的对象文件。

2.2.3 元对象编译器限制

元对象编译器并不展开 #include 或者#define，而只是简单地忽略所遇到的所有预处理程序。并且无法处理所有的 C++语法，主要的问题是类模板不能含有信号和槽。下面是一个错误的例子：

```
class SomeTemplate<int> : public QFrame {
   Q_OBJECT
   ...
signals:
   void bugInMocDetected( int );
};
```

元对象编译器的限制说明如下。

（1）多重继承把 QObject 的子类作为第 1 个父类

如果使用了多重继承，元对象编译器认为第 1 个继承类是 QObject 的子类。这是因为元对象编译器并不展开#include 或#define，它无法发现基类中哪个是 QObject。例如：

```
class SomeClass : public QObject, public OtherClass {
   ...
};
```

（2）函数指针不能作为信号或槽的参数

下面是一个不合法的语法的例子：

```
class SomeClass : public QObject {
   Q_OBJECT
   ...
public slots:
   // 不合法的
   void apply( void (*apply)(List *, void *), char * );
};
```

可以如同下面这样纠正这个错误：

```
typedef void (*ApplyFunctionType)( List *, void * );

class SomeClass : public QObject {
   Q_OBJECT
   ...
public slots:
   void apply( ApplyFunctionType, char * );
};
```

（3）不能把友声明 friend 放在信号或者槽的声明部分

通常情况下，友声明 friend不能放在信号或者槽的声明部分，把它们替换到 private、protect或 public 部分中。如下是一个不合法的例子：

```
class SomeClass : public QObject {
   Q_OBJECT
```

```
    ...
signals:
    friend class ClassTemplate<char>; // 错的
};
```

(4) 信号和槽不能被升级

把继承的成员函数升级为公有状态这一个 C++特征对信号和槽并不适用，下面是一个不合法的例子：

```
class Whatever : public QButtonGroup {
    ...
public slots:
    void QButtonGroup::buttonPressed; // 错的，槽是保护的
    ...
};
```

(5) 宏不能用在信号和槽的参数中

因为元对象编译器不能展开#define，所以在信号和槽中类型宏作为一个参数不能工作，下面是一个不合法的例子：

```
#ifdef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif

class Whatever : public QObject {
    ...
signals:
    void someSignal( SIGNEDNESS(int) );
    ...
};
```

(6) 嵌套类不能放在信号部分或者槽部分，也不能含有信号和槽

下面是一个例子：

```
class A {
  Q_OBJECT
public:
  class B {
  public slots:   // 错的
      void b();
      ...
  };
signals:
  class B {       // 错的
    void b();
    ...
  };
};
```

(7) 构造函数不能用于信号或槽的声明部分

下面是一个错误的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
public slots:
    SomeClass( QObject *parent, const char *name )
        : QObject( parent, name ) { } // 错的
    ...
};
```

（8）属性的声明应该 public 声明部分之前

如果在 public 之后声明属性，元对象编译器将不能找到函数或解析这个类型，下面是一个错误的例子：

```
class SomeClass : public QObject {
   Q_OBJECT
public:
   ...
   Q_PROPERTY( Priority priority READ priority WRITE setPriority ) // 错的
   Q_ENUMS( Priority ) // 错的
   enum Priority { High, Low, VeryHigh, VeryLow };
   void setPriority( Priority );
   Priority priority() const;
   ...
};
```

这个例子纠正错误后列出如下：

```
class SomeClass : public QObject {
   Q_OBJECT
   Q_PROPERTY( Priority priority READ priority WRITE setPriority )
   Q_ENUMS( Priority )
public:
   ...
   enum Priority { High, Low, VeryHigh, VeryLow };
   void setPriority( Priority );
   Priority priority() const;
   ...
};
```

2.2.4 属 性

Qt 的属性也基于元对象系统，在类声明中用宏 Q_PROPERTY 来声明。属性只能在继承于 QObject 的子类中声明，宏 Q_OVERRIDE 用来覆盖一些子类中由继承得到的属性，属性也是一个类的成员。

元对象系统中设置属性和得到属性的成员函数列出如下。

（1）QObject::setProperty()：允许控制类中那些在编译时不可用的属性。

（2）QMetaObject::propertyNames() ：返回所有可用属性的名称。

（3）QMetaObject::property() ：返回一个指定属性的属性数据，即一个 QMetaProperty 对象。

下面两个设置函数是等效的：

```
// QButton *b 和 QObject *o 指向同一个按钮时
b->setDown( TRUE );
o->setProperty( "down", TRUE );
```

示例：使用元对象系统的属性

下面的 MyClass 类用来设置和得到优先级，但还不能使用元对象系统的属性。 MyClass 类列出如下：

```
class MyClass : public QObject
{
   Q_OBJECT
public:
   MyClass( QObject * parent=0, const char * name=0 );
   ~MyClass();

   enum Priority { High, Low, VeryHigh, VeryLow };
```

```
    void setPriority( Priority );
    Priority priority() const;
};
```

为了使用元对象的属性系统，必须用宏 Q_PROPERTY 来声明属性，其语法如下：

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
          [RESET resetFunction] [DESIGNABLE bool]
          [SCRIPTABLE bool] [STORED bool] )
```

宏 Q_PROPERTY 说明如下。

（1）type name：属性的类型及名字，它可以是一个 QVariant 支持的类型或者在类中已经定义的枚举类型。枚举类型必须使用 Q_ENUMS宏来进行注册。

（2）READ getFunction：表示用于读取属性的函数是 getFunction。

（3）WRITE setFunction ：表示用于写（或设置）属性的函数是 setFunction。

（4）RESET resetFunction 表示用函数 resetFunction设置属性为默认状态（可能和初始状态不同），这个函数必须返回 void 并且不带有参数。

（5）DESIGNABLE bool：声明这个属性是否适合被一个图形用户界面设计工具修改， bool 默认为 TRUE 说明这个属性可写；否则说明不能被图形用户界面设计工具修改。

（6）SCRIPTABLE bool：声明这个属性是否适合被一个脚本引擎访问， bool 默认为 TRUE，说明可以被访问。

（7）STORED bool ：声明这个属性的值是否必须作为一个存储的对象状态而被记住， STORED 只对可写的属性有意义，默认为 TRUE。

这样，修改为使用了宏 Q_PROPERTY 的 MyClass 类列出如下：

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

当枚举数据可以被同时被读或写时，必须使用 Q_SETS 来注册这个枚举类型。

宏 Q_CLASSINFO 可以用来把名称/值这样一套属性添加到一个类的元对象中，例如：

```
Q_CLASSINFO( "Version", "3.0.0" )
```

和其他元数据一样，类信息在运行时可以通过元对象 QMetaObject::classInfo() 访问。

2.3 QObject 类

QObject类从 Qt 类继承，是所有 Qt 对象的基类，Qt 类含有全局需要的各种枚举类型及类型的定义。通常情况下，用户不需要关心这个类，因为它是 QObject 及很少几个类的基类。例如，它定义了枚举类型 ButtonState 说明鼠标按钮的状态，并且定义了枚举类型 WidgetState 说明窗口部件的状态等。

2.3.1 对象树

QObject 类是所有 Qt 对象的基类，是 Qt 对象模型的核心。通过 QObject 对象可组织为对象树，QObject 类提供了对对象树进行访问需要的各种成员函数。当创建一个对象时，这个对象的父对象自动调用函数 insertChild()将这个对象插入到父对象的孩子对象链表中。而调用函数 children()可得到孩子对象链表，调用函数 objectTrees()可得到对象树根的所有对象的链表，调用函数 queryList 可查询得到符合查询条件的对象的链表。

对象链表是通过 QObjectList 类实现的，QObject 对象树是一个静态的 QObjectList 类对象 object_trees。object_trees 链表中存有所有的对象指针，通过 object_trees 链表可查询到所有的对象。QObject 对象树的层次图如图 2-3 所示。

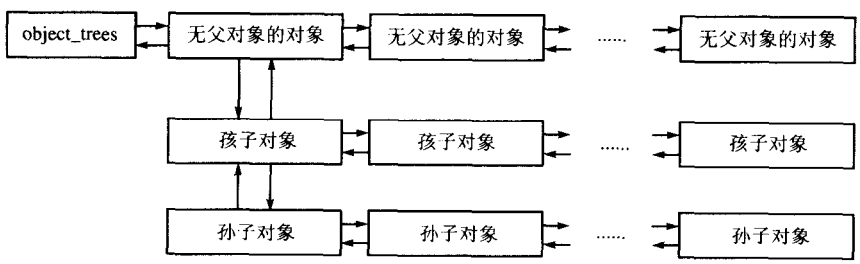


图 2-3 QObject 对象树的层次

从图中可见，QObject 对象是分层管理的，顶层链表链接的是无父对象的对象，一般是顶层窗口。第 2 层链表是无父对象的对象的孩子链表，依次类推。object_trees 对象树与文件系统的文件组织方式类似，这种树型分层结构加快了对象的查找速度。

QObject 对象链表类 QObjectList 从类 QPtrList 继承，类 QObjectList 的定义列出如下：

```
static QObjectList* object_trees = 0;  //整个对象链表
class Q_EXPORT QObjectList : public QPtrList<QObject>
{
public:
   QObjectList() : QPtrList<QObject>() {}
   QObjectList( const QObjectList &list ) : QPtrList<QObject>(list) {}
  ~QObjectList() { clear(); }
   QObjectList &operator=(const QObjectList &list)
     { return (QObjectList&)QPtrList<QObject>::operator=(list); }
};
```

QObject 类构造函数将本对象加到对象树中，列出如下：

```
QObject::QObject( QObject *parent, const char *name )
   :
   isSignal( FALSE ),              //不是一个信号对象
   isWidget( FALSE ),              //不是一个窗口部件对象
   pendTimer( FALSE ),             //还没有定时器
   blockSig( FALSE ),                   //不阻塞信号
   wasDeleted( FALSE ),                 //监控两次以上删除的标识
   isTree( FALSE ),                //不是树，即不在树顶层链表中
   objname( name ? qstrdup(name) : 0 ),          //设置对象名
   parentObj( 0 ),                 // 不是父对象，它被函数 insertChild()设置
   childObjects( 0 ),                   //不是孩子对象
   connections( 0 ),                    //还没有连接
   senderObjects( 0 ),                  //还没有信号连接
   eventFilters( 0 ),                   //还没安装过滤器
```

```
    postedEvents( 0 ),                    //      还没有事件传递
    d( 0 )
{
    if ( !metaObj )                    // 创建对象字典
      (void) staticMetaObject();

    if ( parent ) {//如果父对象存在
    //插这个对象对父对象的孩子对象链表中，如果孩子对象不存在，创建孩子对象链表
      parent->insertChild( this );
    } else {// 父对象不存在,插入到对象树顶层链表中
      insert_tree( this );
      isTree = TRUE;
    }
}
```

下面是一个查询对象的例子，其中主要使用了 QObject 类的函数 queryList，这个例子查询得到所有的 QButton 类对象的链表。如果对象 obj 是链表中的 QButton 类对象，将按钮设置为失效状态。样例代码如下：

```
 QObjectList *l = topLevelWidget()->queryList( "QButton" );  //查询得到 QButton 类所有对象
QObjectListIt it( *l );
QObject *obj;

while ( (obj = it.current()) != 0 ) { // 遍历所有的 QButton 类对象
    //对于每个找到的对象
    ++it;
    ((QButton*)obj)->setEnabled( FALSE );
}
delete l; // 删除链表 l，并未删除对象 obj
```

2.3.2 事件处理过程

在 Qt 中，一个事件是继承自 QEvent 的对象，事件通过调用 QObject::event()被发送到继承自 QObject 的对象。事件发送表明一个事件已经产生，用 QEvent 表达这个事件，且 QObject 需要做出回应。多数事件针对 Qwidget 及其子类，此外还有些与图形无关的重要事件，如套接字激活，它是 QSocketNotifier 使用的事件。

一些事件来自窗口系统，如 QmouseEvent。一些来自其他处，如 QTimerEvent，还有一些来自应用程序。Qt 一视同仁，应用程序可以如同 Qt 自己的事件循环所采用的方式发送事件。

大多数事件类型有特定的类，常用的有 QResizeEvent、QPaintEvent、QMouseEvent、QKeyEvent 和 QcloseEvent，还有很多其他事件类。每个类派生自 QEvent 且添加事件特定的函数，例如 QResizeEvent。在 QResizeEvent 中，添加了 QResizeEvent::size()和 QResizeEvent::oldSize()。

有些类支持多种事件类型，如 QMouseEvent 支持鼠标移动、按压、粘滞按压、拖动、点击及右按压等。

因为程序需要以多变和复杂的方式返回，因此 Qt 的事件分发机制是灵活的。正常发送一个事件的方法是通过调用一个虚拟函数，例如：QPaintEvent 通过调用 QWidget::paintEvent() 被分发。

1．创建用户事件

为创建一个自定义类型的事件，需要定义一个事件号，其值必须大于 QEvent::User。为了传递有关用户自定义事件的特性，自定义的事件可能需要从 QCustomEvent 类继承。

示例 1：用户事件类的编写

下面列出了一个用户事件类 TEST_Event 的示例，编写用户事件类的方法是首先定义一个事件号。

然后实现用户事件类，应用程序将把用户事件类与 Qt 的事件类同等处理。

用户事件类 TEST Event 的头文件 test event.h 列出如下：

```
#include <qevent.h>
#include <qstring.h>

#define REFRESHUI_EVENT QEvent::User+2
class TEST_Event: public QCustomEvent
{
public:
   TEST_Event();
};
```

用户事件类 TEST Event 的实现文件 test_event.cpp 列出如下：

```
#include <test_event.h>
TEST_Event::TEST_Event():QCustomEvent(REFRESHUI_EVENT )
{
}
```

2. 事件发送

许多应用程序都要创建并发送自己的事件，为此需要创建一个相应的事件类的对象，然后调用 QApplication::sendEvent()或者 QApplication::postEvent()发送事件。对用户事件来说，还需要用户有事件对应的操作函数；对于 Qt 事件来说，Qt 中已实现了事件对应的操作函数。

示例 2：使用用户事件类

在 test_engine.cpp 文件中，函数 refreshUI 发送事件到主窗口，要求刷新主窗口界面。在 main_window.cpp 文件中，主窗口类 MainWindow_UI 重载了 customEvent 函数，实现了事件需求的操作，即刷新主窗口界面。

在 test_engine.cpp 文件中，函数 refreshUI 实现代码列出如下：

```
void Engine_View::refreshUI()
{
// pmainWindowGui 是应用程序主窗口类 MainWindow_UI 的实例
qApp->postEvent( pmainWindowGui, new TEST_Event() );  //发送事件
qApp->wakeUpGuiThread();  //唤醒 Gui 线程
}
```

在 main_window.cpp 文件中，函数 customEvent 的实现代码列出如下：

```
void MainWindow_UI::customEvent( QCustomEvent * e)
{

   if ( e->type() == REFRESHUI_EVENT )
   {
     进行主窗口刷新操作
   }
}
```

在应用程序的基类 QApplication 中有事件处理函数的实现，其中事件的发送函数说明如下：

```
bool QApplication::sendEvent ( QObject * receiver,QEvent * event ) [static]
```

sendEvent() 立即发送事件给接收对象 receiver，当其返回时，（事件过滤器和）对象已经处理过事件。对于很多事件类，可以通过调用 isAccepted()函数知道该事件能否被处理者所接受或者拒绝。

```
void QApplication::postEvent ( QObject * receiver, QEvent * event ) [static]
```

postEvent()投寄事件到一个队列，以便能延迟分发。在下次 Qt 的主事件循环运行时，它分发全部事件。还可进行一些优化，例如，若有数个 resize 事件，则就被压缩成一个。对于 paint 事件同样如此，QWidget::update() 调用了 postEvent()。postEvent()投寄事件可以避免屏幕因多次重绘而闪烁，同时还加快了运行速度。

postEvent()在对象初始化期间常常被使用，因为在对象完成初始化后，投送的消息会被很快派发。

```
bool QApplication::notify ( QObject * receiver, QEvent * e ) [virtual]
```

发送事件到接收对象者 receiver，返回值是从 receiver 的事件处理函数中返回的值。对于某一类型的事件（如鼠标和键盘事件）来说，如果接收者对事件不感兴趣（如返回 FALSE），事件将被传播到 receiver 的父类。如果父类不感兴趣，则一直向上级传递，直到顶层的 object 类。

3. 事件处理

有 5 种不同的处理事件的方法，列出如下。

（1）重载函数 QApplication::notify()可提供有效的事件控制，但仅能在派生于 QApplication 的类中重实现这个函数。

（2）在 qApp（是 QApplication 的全局实例）中实现事件过滤，这样的一个事件过滤器能为所有的 widget 处理所有的事件，而且可以有超过一个全局应用程序的事件过滤器。如鼠标事件的全局事件过滤器设置了鼠标跟踪使能，则鼠标移动事件对于所有 widget 有效。

（3）重载 QObject::event()（在 QWidget 类中）在任何 widget 特定的事件过滤器之前可看到所有事件，其函数声明列出如下：

```
bool QObject::event ( QEvent * e ) [virtual]
```

这个虚函数接收一个对象的事件，如果事件被识别并被处理，将返回 TRUE。这个函数能被用来重实现一个对象的行为。

（4）在对象上安装事件过滤器。

（5）重载 Qt 基类事件处理函数。

当用户发现 Qt 基类的事件处理函数不能满足需要时，可以在用户类中重载这些函数。对于特定的 Qt 事件，可以重载特定的事件函数，如重载 paintEvent() 及 mousePressEvent()等函数。如果想修改多个 Qt 事件处理函数，可以通过重载 QObject::event()来实现。

示例 3：重载 QObject::event()

下面是重载 QObject::event() 函数的例子，它执行特定的 tab 键处理，还处理用户事件：

```
bool MyClass:event( QEvent * e )
{
  if ( e->type() == QEvent::KeyPress )
  {
     QKeyEvent * ke = (QKeyEvent*) e;
     if ( ke->key() == Key_Tab )
     {
        // 这里是特定的tab处理
        k->accept();
        return TRUE;
     }
  }
  else if ( e->type() >= QEvent::User )
  {
```

```
        QCustomEvent * c = (QCustomEvent*) e;
        // 这里是自定义事件处理
        return TRUE;
    }
    QWidget::event( e );
}
```

2.3.3 事件运行机制

当应用程序的 main 函数调用 qApp->exec() 时，应用程序进入 Qt 的主事件循环。Qt 的主事件循环从事件队列中取出本窗口及系统事件，把它们转换为 QEvents 并使用函数 QApplication::notify 发送转换的事件给相应的对象 QObjects。同时，还处理控制台 tty 的信号、QWSserver 服务器的事件，并将来自 socket 的消息转换为事件进行分发。这些事件的处理工作在函数 QEventLoop::processEvents (flags) 中完成。

Qt 的主事件循环函数 QApplication::exec() 的调用层次如图 2-4 所示，从函数的调用层次可看出事件的分发处理流程。这里没有给出源代码分析，读者可参考 Qt 3.4 源代码。

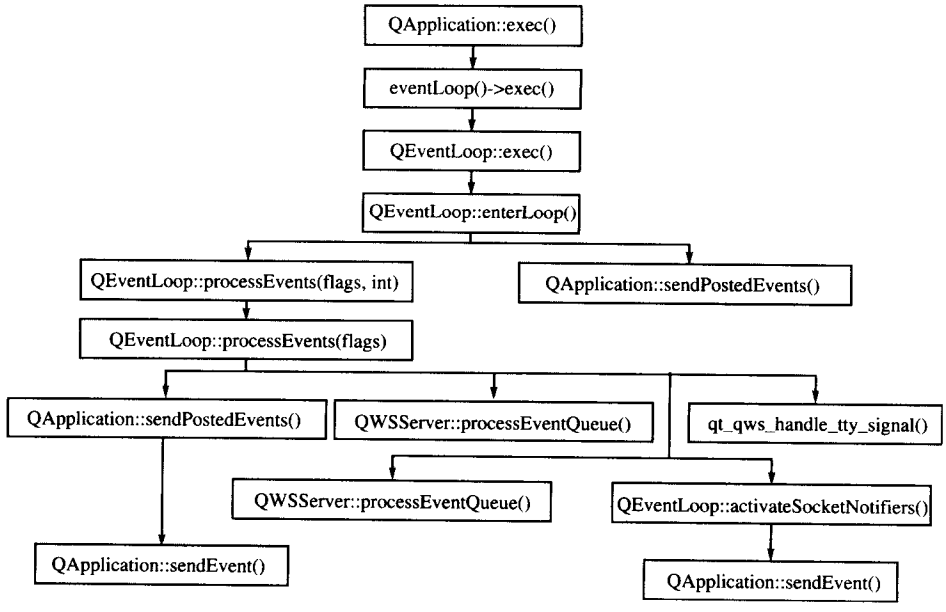


图 2-4 函数 QApplication::exec() 的调用层次

通常事件来自于窗口系统（用 spontaneous()函数检查时返回 TRUE），也可能来自使用 Application::sendEvent() 和 QApplication::postEvent()手动发送的事件（用 spontaneous()函数检查时返回 FALSE）。

QObjects 通过调用其 QObject::event()接收事件，这个函数也可被子类重载来处理事件，最典型的重载是 QWidget::event()。

2.3.4 事件过滤器

一个事件过滤器是一个能接收所有发送到这个对象上的事件的对象，并且能停止或转发到这个对象上的事件。事件过滤器通过 eventFilter()函数来接收事件，如果这个事件应该被过滤（如停止事件等），eventFilter()函数返回 TRUE；否则返回 FALSE。如果多个事件过滤器被安装在一个对象上，最后安装的过滤器将被激活。

QObject类还提供了事件过滤器的安装，QObject 类与事件过滤相关的几个成员函数说明如下。

（1）bool QObject::event (QEvent * e) [virtual]：接收一个对象的事件，如果事件被识别并处理，返回 TRUE。这个函数能被重载来定制一个对象的行为。

（2）bool QObject::eventFilter (QObject * watched, QEvent * e) [virtual]：如果一个对象上已安装了事件过滤器，eventFilter 函数将被用来过滤事件。在这个函数的重载中，如果想过滤事件 e（如使其停止不再被处理），则返回 TRUE；否则返回 FALSE。

（3）void QObject::installEventFilter (const QObject * filterObj)：在对象 filterObj 上安装事件过滤器。

下面是一个事件过滤器的使用样例，MyMainWindow 类在本对象上安装了事件过滤器，事件过滤处理函数 eventFilter 被重载用来在 textEdit 对象上处理 KeyPress 事件。未处理的事件被传递到基类的 eventFilter() 函数中，因为基类也可能因为内部事件处理的原因已重载了 eventFilter() 函数。如果在这个函数中删除了接收者对象，则确认返回 TRUE；否则 Qt 将向前传递事件到删除的对象中，则程序将崩溃。

```
class MyMainWindow : public QMainWindow
{
public:
   MyMainWindow( QWidget *parent = 0, const char *name = 0 );

protected:
   bool eventFilter( QObject *obj, QEvent *ev );

private:
   QTextEdit *textEdit;
};

MyMainWindow::MyMainWindow( QWidget *parent, const char *name )
     : QMainWindow( parent, name )
{
     textEdit = new QTextEdit( this );
     setCentralWidget( textEdit );
     textEdit->installEventFilter( this ); //在本对象上安装事件过滤器
}

bool MyMainWindow::eventFilter( QObject *obj, QEvent *ev )
{
     if ( obj == textEdit ) { //  过滤的对象
       if ( e->type() == QEvent::KeyPress ) {// 过滤的事件
          qDebug( "Ate key press %d", k->key() );
          return TRUE; //已处理事件，必须返回 TRUE，这样系统不会对这个事件做第 2 次处理
       } else {
          return FALSE;
       }
     } else {
       // 传递事件到父类
       return QMainWindow::eventFilter( obj, ev );
    }
}
```

2.3.5 定时器

使用定时器有两种方法，一种是使用 QTimer 类；另一种是使用 QObject 类的定时器。定时器的精确度依赖于操作系统和硬件，大多数平台支持 20 ms 的精确度。

1. QObject 类的定时器

Qobject 是所有 Qt 对象的基类，它提供了一个基本的定时器。通过 QObject::startTimer()，可以把一个以毫秒为单位的时间间隔作为参数来开始定时器，这个函数返回一个惟一的整数定时器的标识符。

这个定时器现在就会在每一个时间间隔“触发”，直到明确地使用这个定时器的标识符来调用QObject::killTimer()结束。

应用程序在 main 函数中通过调用 QApplication::exec()来开始事件循环。当定时器触发时，应用程序会发送一个 QtimerEvent。在事件循环中，处理器按照事件队列顺序来处理定时器事件。当处理器正忙于其他事件处理时，定时器就不能立即触发。

QObject 类还提供定时器的功能，与定时器相关的成员函数有 startTimer()、timerEvent()、killTimer()和 killTimers()。QObject 基类中的 startTimer 和 timerEvent 原型及说明如下：

```
int QObject::startTimer ( int interval )
```

开始一个定时器并返回定时器 ID，如果不能开始一个定时器，将返回 0。定时器开始后，每隔 interval 毫秒间隔将触发一次超时事件，直到 killTimer()或 killTimers()被调用来删除定时器。如果 interval 为 0，那么定时器事件每次发生时没有窗口系统事件处理。

```
void QObject::timerEvent ( QTimerEvent * ) [virtual protected]
```

虚拟函数 timerEvent 被重载来实现用户的超时事件处理函数。如果有多个定时器在运行，QTimerEvent::timerId() 被用来查找定时器的文件。

当定时器事件发生时，虚函数 timerEvent()随着 QTimerEvent 事件参数类一起被调用，重载这个函数可以获得定时器事件。

定时器的用法样例如下：

```
class MyObject : public QObject
   {
     Q_OBJECT
   public:
     MyObject( QObject *parent = 0, const char *name = 0 );

   protected:
     void timerEvent( QTimerEvent * );
   };

   MyObject::MyObject( QObject *parent, const char *name )
     : QObject( parent, name )
   {
     startTimer( 50 );    // 50 ms 定时器
     startTimer( 1000 );  // 1 s 定时器
     startTimer( 60000 ); // 1 分钟定时器
   }

void MyObject::timerEvent( QTimerEvent *e )  //重载 timerEvent 函数
{
   qDebug( "timer event, id %d", e->timerId() );
}

void MyObject::stopTimer()
{
   killTimer( showDateTimer );
   showDateTimer = -1;
}
```

2. 定时器类 QTimer

定时器类 QTimer 提供当定时器触发的时候发射一个信号的定时器，它提供只触发一次的超时事件，通常的使用方法如下：

```
QTimer * testtimer = new QTimer( this );  //   创建定时器
```

```
connect(testtimer, SIGNAL(timeout()),
        this, SLOT(updateCaption()) );// 将定时器超时信号与槽（功能函数）连接起来
testtimer ->start( 1000 );  //开始运行定时器，定时时间间隔为 1000 ms
```

testtimer定时器被作为这个窗口部件的子类，这样当这个窗口部件被删除时，定时器也会被删除。

QTimer还提供了一个简单的只有一次定时的函数 singleShot，例如，一个定时器在 100 ms 后触发处理函数 animateTimeout 并且只触发一次。代码如下：

```
QTimer::singleShot( 100, this, SLOT(animateTimeout()) );
```

2.3.6 连接函数 connect

QObject 类提供了信号与槽的连接函数 connect()和断开连接函数 disconnect()，这两个函数的用法在前面已经详细说明。

2.3.7 字符串翻译函数

QObject 类还提供了字符串的翻译函数 tr()和 trUtf8()，说明如下。

（1）QString QObject::tr(const char * sourceText, const char * comment) const：

返回字符串 sourceText 的翻译，如果没有合适的翻译版本，则返回原字符串 sourceText 。参数 comment 是翻译的上下文，是对字符串的补充说明，如说明属于哪个类。它可用来辅助标识字符串，这样可做到相同字符串要求不同的翻译。

（2）QString QObject::trUtf8 (const char * sourceText, const char * comment) const：

返回字符串 sourceText 的翻译，如果没有合适的翻译版本，则返回字符串 QString:: fromUtf8 (sourceText)，其他与函数 tr()相同。

2.4 Qt 国际化

软件的国际化是指软件支持多国语言，能适应不同用户的语言、输入方法、字符编码，甚至表达习惯等。软件的国际化使软件能被多个国家的人使用。

2.4.1 软件中字符串国际化方法

软件可以按下面的几种方法实现国际化。

1. 对用于 Gui 界面的字符串使用 QString

QString 内部使用了 Unicode 编码，Unicode 编码包括了世界上每种语言的字符编码。在 Qt 的基类中，有关字符串的函数大多数使用了 QString 作为参数，这样减小了 char*到 QString 的转换的时间开销。

另外，还有 QCString 和 QChar，它们的用法类似于传统 C 中的 const char*和 char。通过使用 QString、QCString 和 QChar 类，开发者不会注意到正在使用 Unicode。

2. 对所有文字形式的文本使用 tr()

当程序需要为用户显示字符串时，要确保其已被 QApplication::translate()函数处理，为此只需要使用 QObject::tr()。例如，假设 LoginWidget 是 QWidget 的一个子类：

```
LoginWidget::LoginWidget()
{
   QLabel *label = new QLabel(tr("Password:"), this );
   ...
}
```

这样就解决了大部分可能要写的用户可见的字符串。如果这些被引用的字符串不是在 QObject 子类的成员函数中，可以使用一个适当的类的 tr()函数，或者直接使用 QApplication::translate()函数，使用这两个函数的例子如下：

```
void some_global_function( LoginWidget *logwid )
{
   QLabel *label = new QLabel( LoginWidget::tr("Password:"), logwid );
}

void same_global_function( LoginWidget *logwid )
{
   QLabel *label = new QLabel(
            qApp->translate("LoginWidget", "Password:"),
            logwid );
}
```

lupdate 工具会自动为每个源文本提供一个上下文，它是装有 tr()调用的类的类名，足够用来处理相同字符串需要不同翻译大多数情况。但有时候，需要更多的信息来惟一地标识源文本。

示例：用上下文来惟一标识字符串

一个装有两个按钮的对话框，每个按钮上有“ Enabled”字符串。但翻译需要有差异，这时需要用上下文来惟一标识字符串。这两个按钮字符翻译的上下文，分别是“ Color frame ”和 “Hue frame”。这个例子列出如下：

```
rbc = new QRadioButton( tr( "Enabled", "Color frame" ), this);
rbh = new QRadioButton( tr( "Enabled", "Hue frame" ), this);
```

3. 翻译函数外的文本

如果需要翻译函数外的文本，有两个宏可以使用，即 QT_TR_NOOP() 和 QT_TRANSLATE_NOOP()。它们仅仅为文本做出标签，以便于被 lupdate工具提取。使用 QT_TR_NOOP()的例子如下：

```
QString FriendlyConversation::greeting( int greet_type )
{
   static const char* greeting_strings[] = {
           QT_TR_NOOP( "Hello" ),
           QT_TR_NOOP( "Goodbye" )
   };
   return tr( greeting_strings[greet_type] );
}
```

使用 QT_TRANSLATE_NOOP() 的例子如下：

```
static const char* greeting_strings[] = {
   QT_TRANSLATE_NOOP( "FriendlyConversation", "Hello" ),
   QT_TRANSLATE_NOOP( "FriendlyConversation", "Goodbye" )
};

QString FriendlyConversation::greeting( int greet_type )
{
   return tr( greeting_strings[greet_type] );
}
```

```
QString global_greeting( int greet_type )
{
   return qApp->translate( "FriendlyConversation",
                           greeting_strings[greet_type] );
}
```

如果使用了宏定义 QT_NO_CAST_ASCII 编译自己的软件，就会关闭从 const char* 到 QString 的自动转换，并且很可能会得到错误的字符串。

如果源码语言使用 Latin-1 之外的字符集，则会发现 QObject::trUtf8() 比 QObject::tr() 更好用。因为 tr() 依赖于 QApplication::defaultCodec() ，这使它比 QObject::trUtf8() 更脆弱。

对于加速键值（Accelerator value），例如 Ctrl+Q 或者 Alt+F，有时需要翻译者重载它，这时可使用 QKeySequence()。如果应用为“Quit”直接编码（hardcode）为 CTRL+Key_Q，则翻译者不能重载它。正确的用法如下：

```
QPopupMenu *file = new QPopupMenu( this );
file->insertItem( tr("&Quit"), this, SLOT(quit()),
                  QKeySequence(tr("Ctrl+Q", "File|Quit")) );
```

4. 对字符串参数使用 QString::arg()

对于国际化文本，在字符串中使用类似 printf() 风格的插入参数不是好的选择，因为有时候有必要在翻译时改变参数的顺序。 QString::arg() 函数为参数替换提供了一种简单的途径，下面是使用 QString::arg() 的例子：

```
void FileCopier::showProgress( int done, int total,
                               const QString& current_file )
{
   label.setText( tr("%1 of %2 files copied.\nCopying: %3")
                  .arg(done)
                  .arg(total)
                  .arg(current_file) );
}
```

2.4.2 创建译本

在程序代码中使用 tr() 之后，即可开始制作程序中用户可见文本的译本了。 Qt提供了 Qt 语言学家（Qt Linguist）工具、lupdate 和 lrelease 来进行翻译处理。其中 Qt Linguist 是图形界面工具，用户可以使用它将字符串对应的翻译填入 .ts 文件中。

在运行 lupdate 之前，应该准备一个项目文件（.pro 文件）。下面是一个项目文件名为 myproject.pro 的例子：

```
HEADERS       = funnydialog.h \
                wackywidget.h
SOURCES       = funnydialog.cpp \
                main.cpp \
                wackywidget.cpp
FORMS         = fancybox.ui
TRANSLATIONS    = superapp_dk.ts \
                superapp_fi.ts \
                superapp_no.ts \
                superapp_se.ts
```

当运行 lupdate 或者 lrelease 时，必须以命令行参数给出项目文件的名称。例如， lupdate myproject.pro。在本例中支持 4 种语言，即 Danish、Finnish、Norwegian 和 Swedish。如果使用 qmake（或者 tmake），则一般不需要为 lupdate 附加项目文件。而只要添加上 TRANSLATIONS 项，qmake

项目文件就会正常工作。

Qt应用的翻译过程分为如下 3 步。

（1）运行 lupdate，即使用命令 lupate project-file –ts ts-files 提取 Qt 应用的 C++源代码中的可翻译文本，会产生一个翻译信息文件（.ts文件）。lupdate 能识别出 tr()结构和上面描述的 QT_*_NOOP 宏。lupdate 读一个.pro 项目文件，并产生或更新列出在项目文件中的.ts 翻译源文件。

（2）使用 Qt语言学家（Qt Linguist）翻译工具软件打开.ts 文件，然后人工翻译.ts 文件中源文本的字符串，将字符串的译文使用 Qt Linguist 工具加到.ts 文件中。.ts 文件是 XML 格式，也可以手工编辑它们。

（3）运行 lrelease，即使用命令 lrelease ts-files –qm qm-file 从.ts 文件中得到只适用于最后使用的轻量级的信息文件（.qm 文件）。你可以把.ts 文件看成“源文件”，把.qm 文件看成“目标文件”。翻译者编辑的是.ts 文件，但是应用程序的用户只需要.qm 文件。这两种文件都是平台和地区（locale）无关的，应将对应用的每个发表版本重复上述 3 步。lupdate 工具会尽力重用以前发表版本的译文。

在应用中，必须使用 QTranslator::load()来装载对应用户语言的译文文件，然后使用 QApplication::installTranslator()来安装它们。

如果一直使用以前的 Qt 工具（findtr、msg2qm 和 mergetr），可以使用 qm2ts 来转换以前的.qm 文件。

虽然这些工具提供了生成.qm 文件的方便途径，可在任何能编写.qm 文件的系统中使用。但是可以编写一个应用，以利用 QTranslator::insert() 把译文加入到 QTranslator 中，然后利用 QTranslator::save()写出一个.qm 文件。用这种方法可以从任何你选择的源文件中产生译文。

Qt 本身包含有大约 400 个也需要翻译为目标语言的字符串。在 $QTDIR/translations 下可以找到 French 和 German 的译文文件，也可以作为翻译为其他语言的模板。

示例：装载译文文件

下面的例子说明了如何装载译文文件，其中应用程序会根据不同的本地语言来安装相应的译文文件；

```
int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    /*装载 Qt 翻译文件*/
QTranslator qt( 0 );  //装载没有父类的 QTranslator 对象
    //根据本地语言选择译文文件（.qm 文件）
    qt.load( QString( "qt_" ) + QTextCodec::locale(), "." );
    app.installTranslator( &qt );

    //给应用程序字符串装载翻译文件
    QTranslator myapp( 0 );
    myapp.load( QString( "myapp_" ) + QTextCodec::locale(), "." );
    app.installTranslator( &myapp );
    ...
    return app.exec();
}
```

2.4.3 编码支持

当应用启动时，机器的地区（locale）决定了处理 8 位文本数据方法（如字体选择、文本显示、 8 位文本 I/O和字符输入所用的 8 位编码）。

应用程序中有时会需要不同于默认的本地 8 位编码，QTextCodec 类和 QTextStream 类中的函数支持很多对于用户数据的输入输出编码。例如，一个 Cyrillic KOI8-R locale（俄罗斯的事实标准 locale）

的应用可能需要以 ISO 8859-5 编码输出 Cyrillic，这样的代码可能会是：

```
QString string = ...; // 一些Unicode编码的文本

QTextCodec* codec = QTextCodec::codecForName( "ISO 8859-5" );  //使用指定的编码
QCString encoded_string = codec->fromUnicode( string ); //得到指定编码的字符串
```

把 Unicode 转换为本地 8位编码的快捷办法是 QString 的 local8Bit()方法，它返回 8位数据。另一种有用的快捷办法是 utf8()方法，它以 8位的 UTF-8编码返回文本。UTF-8编码的好处是如果 Unicode 完全是 US-ASCII 的话，它可以完全保留 Unicode 信息，而看起来又是一般的 US-ASCII。

Qt 提供完整的 Unicode 支持，包括输入法、字体、剪贴板、拖放和文件名。文件 I/O 默认为 Latin-1，在 QTextStream 中带有 Unicode 选项。Unicode（UTF16 或者 UTF8）是包含了所有语言的字符编码方法，使用 Unicode 编码的软件可以最大程度地提高各国用户之间的文本兼容性。然而有时在同一语言的群体内，本地编码标准也许更为合适，这个编码就是由 QTextCodec::codecForLocale()所返回的编码方法。开发者还可以使用 QTextCodec::loadCharmapFile() 函数构造一个数据驱动的编码器。

Qt 所支持的字符编码类及其功能说明如表 2-2 所示。

表 2-2 Qt所支持的字符编码类及其功能说明

类 名	功能及其功能说明
QEucJpCodec	EUC 到 JP 字符集之间的转换
QEucKrCodec	EUC 到 KR 字符集之间的转换
QGb18030Codec	中国 GB18030/GBK/GB2312 编码
QGbkCodec	中国 GBK 编码
QHebrewCodec	Hebrew 编码
QJisCodec	JIS 字符集编码
QSjisCodec	Shift-JIS 字符集编码
QTextCodec	文本编码之间的转换
QTextDecoder	基于状态（State-based）的解码器
QTextEncoder	基于状态的编码器
QTranslator	文本输出的国际化支持
QTranslatorMessage	Translator 的信息及其属性
QTsciiCodec	Tscii 编码

2.5 QMetaObject 元对象类

QMetaObject 类装有有关 Qt 对象的元信息，包括运行时类型信息和属性系统，它负责信号与槽机制。每个在类声明中含有 Q_OBJECT 宏的类都有一个对应的元对象，即 QMetaObject 类对象。元对象是静态的，因此只有一个元对象与这个类对应。

2.5.1 相关的数据结构

相关的数据结构说明如下。

（1）QConnection、QConnectionList 和 QSignalVec 类

QConnection 类是内部使用的类，用于 signal/slot 机制中，而不能直接用于应用程序。 QObject 中

每个连接到外面的信号 siganl 有一个 QConnection 链表。QConnection 类存储并可使用成员函数得到连接到的对象指针、连接的成员名及其类型和参数个数。 QConnection 类构造函数的声明列出如下：

```
QConnection::QConnection( const QObject *object, int member,
                const char *memberName, int memberType )
```

QConnectionList 类是一个 QConnection 对象链表，它从 QPtrList<QConnection>继承,存储各个 QConnection 对象指针。

QSignalVec 类从 QPtrVector<QConnectionList>继承，它是 QConnection 对象链表的数组，将每个信号的 QConnection 对象链表组织到一个数组中。

（2）元数据结构 QMetaData 及类属性结构 QmetaEnum

结构 QMetaData 是信号与槽成员函数的元数据，用于描述信号或槽，列出如下：

```
struct QMetaData
{
   const char *name;                          // 信号或槽的名称
   const QUMethod* method;                    // 信号或槽详细的方法描述
   enum Access { Private, Protected, Public };
   Access access;                             //访问许可
};
typedef const QMetaData QconstMetaData;
```

类 QmemberDict 从 QasciiDict<QconstMetaData >继承,是 QmetaData 的对象链表。

结构 QmetaEnum 是某类属性的描述，用于显示在 Qt Designer 等图形开发工具上，说明如下：

```
struct QmetaEnum
{
   const char *name;              //某类属性名
   uint count;                    //属性项数总数
   struct Item                    // 一个属项，name/value 对
   {
     const char *key;
     int value;
   };
   const Item *items;             //属性项的数组
   bool set;                      //这个类属性是否被看成一个集合
};
```

2.5.2 QMetaObject 对象

每个含有 Q_OBJECT 宏的类都有一个元对象，本节以 Qt 类的元对象 qtMetaObject 为例说明元对象的建立及管理。

函数 staticQtMetaObject 是 QObject 类的成员函数，用来创建 Qt 类对应的元对象 qtMetaObject，qtMetaObject 是静态的 QMetaObject 类型指针。cleanUp_Qt 是静态的全局 QMetaObjectCleanUp 对象函数 staticQtMetaObject 分析如下：

```
// 将"QObject"及 &QObject::staticMetaObject 插入到qt_metaobjects 链表中
static QMetaObjectCleanUp cleanUp_Qt = QMetaObjectCleanUp( "QObject",
                    Object::staticMetaObject );

QMetaObject* QObject::staticQtMetaObject()
{
   static QMetaObject* qtMetaObject = 0; //静态对象指针
   if ( qtMetaObject )
     return qtMetaObject;
```

```
#ifndef QT_NO_PROPERTIES
   //用于显示在Qt Designer工具中的属性对话框表格中的属性项
   static const QMetaEnum::Item enum_0[] = {
     { "AlignLeft", (int) Qt::AlignLeft },
     ...
     { "AlignJustify", (int) Qt::AlignJustify },
     { "WordBreak", (int) Qt::WordBreak }
   };
  ......
   static const QMetaEnum enum_tbl[] = {
//在表格中的属性名、项计数各项是否作为一个集合
     { "Alignment", 10, enum_0, TRUE },
     { "Orientation", 2, enum_1, FALSE },
     { "TextFormat", 4, enum_2, FALSE },
     { "BackgroundMode", 17, enum_3, FALSE },
     { "DateFormat", 3, enum_4, FALSE }
   };
#endif

   qtMetaObject = new QMetaObject( "Qt", 0,
                0, 0,
                0, 0,
#ifndef QT_NO_PROPERTIES
                0, 0,
                enum_tbl, 5,
#endif
                0, 0 );
   cleanUp_Qt.setMetaObject( qtMetaObject );

   return qtMetaObject;
}
```

类 QMetaObject 构造函数将各种元数据存入到 QMetaObject 元对象中，每个类有一个元对象对应。每个元对象使用链表存储槽或信号的元数据，使用私有类 QMetaObjectPrivate 对象存储属性数据。这样，通过类元对象链表，提供信号与槽连接时的数据查找路径。下面分析上面例子中用到的 QMetaObject 类成员函数及 QMetaObjectCleanUp 类成员函数。QMetaObject构造函数分析如下：

```
QMetaObject::QMetaObject( const char *const class_name, QMetaObject *super_class,
                          const QMetaData *const slot_data, int n_slots,//槽的元数据
                          const QMetaData *const signal_data, int n_signals,//信号的元数据
#ifndef QT_NO_PROPERTIES
                          const QMetaProperty *const prop_data, int n_props, //属性元数据
                          const QMetaEnum *const enum_data, int n_enums, //属性项数据
#endif
                          const QClassInfo *const class_info, int n_info )//类信息元数据
{
    classname = class_name;   // 类的名字
    superclass = super_class; //父元对象
    superclassname = superclass ? superclass->className() : 0;
    slotDict = init( slotData = slot_data, n_slots );  //建立槽的元数据链表
    signalDict = init( signalData = signal_data, n_signals ); //建立信号的元数据链表

    d = new QMetaObjectPrivate;
    reserved = 0;
    //将属性数据存入私有对象d中
#ifndef QT_NO_PROPERTIES
    d->propData = prop_data;
    d->numPropData = n_props;
    d->enumData = enum_data;
    d->numEnumData = n_enums;
#endif
    d->classInfo = class_info;
    d->numClassInfo = n_info;
```

```
    signaloffset = superclass ? ( superclass->signalOffset() + superclass->numSignals() ) : 0;
    slotoffset = superclass ? ( superclass->slotOffset() + superclass->numSlots() ) : 0;
#ifndef QT_NO_PROPERTIES
    propertyoffset = superclass ? ( superclass->propertyOffset() + superclass->numProperties() ) : 0;
#endif
}
```

函数 init 创建 QMetaData 对象链表并插入 data 数组中的所有元数据，列出如下：

```
QMemberDict *QMetaObject::init( const QMetaData * data, int n )
{
    if ( n == 0 )                    // nothing, then make no dict
      return 0;
//创建 QMetaData 对象链表 QAsciiDict<QConstMetaData>
    QMemberDict *dict = new QMemberDict( optDictSize(n), TRUE, FALSE );
    Q_CHECK_PTR( dict );
    while ( n-- ) {                  //将 data 数组所有元数据插入到链表中
      dict->insert( data->name, data );
      data++;
    }
    return dict;
}
```

类 QMetaObjectCleanUp 在 moc 产生的 .cpp 文件中用做静态的全局对象，被用来删除由函数 setMetaObject 设置的 QMetaObject 元对象。函数 setMetaObject 设置 QObject 引用到 metaObject，当 metaObject 被销毁时，metaObject 为 NULL。

所有类名及生成的对应静态元对象都存入到全局的链表 qt_metaobjects 中，构造函数 QMetaObjectCleanUp 用来将类名及生成的对应元对象 func 加入到链表 qt_metaobjects 中，列出如下：

```
static QAsciiDict<void> *qt_metaobjects = 0;
static int qt_metaobjects_count = 0;
typedef QMetaObject *(*QtStaticMetaObjectFunction)()
//func 为元对象构造函数
QMetaObjectCleanUp::QMetaObjectCleanUp( const char *mo_name, QtStaticMetaObjectFunction func )
    : metaObject( 0 )
{
#ifdef QT_THREAD_SUPPORT
    QMutexLocker( qt_global_mutexpool ?
                  qt_global_mutexpool->get( &qt_metaobjects ) : 0 );
#endif // QT_THREAD_SUPPORT
    if ( !qt_metaobjects )
      qt_metaobjects = new QAsciiDict<void>( 257 );
    qt_metaobjects->insert( mo_name, (void*)func );

    qt_metaobjects_count++;
}
```

函数 setMetaObject 设置元对象给 QMetaObjectCleanUp 类内部使用的变量 metaObject，列出如下：

```
void QMetaObjectCleanUp::setMetaObject( QMetaObject *&mo )
{
#if defined(QT_CHECK_RANGE)
    if ( metaObject )
      qWarning( "QMetaObjectCleanUp::setMetaObject: Double use of QMetaObjectCleanUp!" );
#endif
    metaObject = &mo;
}
```

2.5.3 Q_OBJECT 宏及 moc 生成代码分析

使用信号与槽的类在类的声明中都必须加上 Q_OBJECT 宏（在 src/kernel/qobjectdefs.h 中），表示使用元对象系统，Q_OBJECT 列出元对象的声明、类名、属性函数及字符串翻译函数等。该宏的定义列出如下：

```
#define Q_OBJECT                                        \
public:                                                 \
    virtual QMetaObject *metaObject() const {           \
        return staticMetaObject();                      \//返回静态的元对象
    }                                                   \
    virtual const char *className() const;              \
    virtual void* qt_cast( const char* );               \
    virtual bool qt_invoke( int, QUObject* );           \  //触发信号
    virtual bool qt_emit( int, QUObject* );             \  //发送信号
    QT_PROP_FUNCTIONS                                   \
    static QMetaObject* staticMetaObject();             \
    QObject* qObject() { return (QObject*)this; }       \
    QT_TR_FUNCTIONS                                     \
private:                                                \
    static QMetaObject *metaObj;


//如果 object 对应的 QMetaObject 对象从 T 的 QMetaObject 对象继承，返回(T)object
template <typename T>
Q_INLINE_TEMPLATES T qt_cast(const QObject *object)
//如果 object 对应的 QMetaObject 对象为((T)0)->staticMetaObject()时，返回 object；否则返回 FALSE
{ return (T)qt_inheritedBy( ((T)0)->staticMetaObject(), object ); }

//属性函数
# define QT_PROP_FUNCTIONS \
    virtual bool qt_property( int id, int f, QVariant* v); \
    static bool qt_static_property( QObject* , int, int, QVariant* );
//翻译函数
#  define QT_TR_FUNCTIONS \
    static QString tr( const char *, const char * = 0 ); \
    static QString trUtf8( const char *, const char * = 0 );
```

元对象编译器 moc 根据类声明中的宏 Q_OBJECT 和 Q_PROPERTY，以及 signal 及 slot 部分的定义生成附加的 C++代码。下面用两个例子进行说明，示例 1 说明类声明在 .h 文件中，moc 编译器生成 moc_*.cpp 附加文件，类声明中有 slot 部分；示例 2 说明类声明在 *.cpp 文件中，moc 编译器生成 .moc 附加文件。

示例 1：QMessageBox 类经对象编译器 moc产生的文件

$QTDIR/src/dialogs/qmessagebox.h 文件是 QMessageBox 类的定义，类声明中含有 Q_OBJECT 宏，经 moc 编译器产生了 $QTDIR/src/.moc/release-shared/moc-qmessagebox.cpp 文件。

QMessageBox 类声明中与元对象相关的代码（在 qmessagebox.h 中）列出如下：

```
class Q_EXPORT QMessageBox : public QDialog
{
    Q_OBJECT
    Q_ENUMS( Icon )
    Q_PROPERTY( QString text READ text WRITE setText )
    Q_PROPERTY( Icon icon READ icon WRITE setIcon )
    Q_PROPERTY( QPixmap iconPixmap READ iconPixmap WRITE setIconPixmap )
    Q_PROPERTY( TextFormat textFormat READ textFormat WRITE setTextFormat )
```

```
public:
   ......
private slots:
   void   buttonClicked();
   ......
}
```

moc 编译器生成的 QMessageBox 类对应的元对象函数（在 moc-qmessagebox.cpp 中）列出如下：

```
#undef QT_NO_COMPAT
#include "../../dialogs/qmessagebox.h"
#include <qmetaobject.h>
#include <qapplication.h>

#include <private/qucomextra_p.h>
...
#include <qvariant.h>
const char *QMessageBox::className() const
{
   return "QMessageBox";
}

QMetaObject *QMessageBox::metaObj = 0;
//将类名及元对象插入到qt_metaobjects链表中
static QMetaObjectCleanUp cleanUp_QMessageBox( "QMessageBox", &QMessageBox::staticMetaObject );

#ifndef QT_NO_TRANSLATION
QString QMessageBox::tr( const char *s, const char *c )  //字符串翻译函数
{
    if ( qApp )
      return qApp->translate( "QMessageBox", s, c, QApplication::DefaultCodec );
    else
      return QString::fromLatin1( s );
}
#ifndef QT_NO_TRANSLATION_UTF8
QString QMessageBox::trUtf8( const char *s, const char *c )  //字符串翻译函数
{
    if ( qApp )
      return qApp->translate( "QMessageBox", s, c, QApplication::UnicodeUTF8 );
    else
      return QString::fromUtf8( s );
}
#endif // QT_NO_TRANSLATION_UTF8

#endif // QT_NO_TRANSLATION
//得到各种元数据，生成元对象
QMetaObject* QMessageBox::staticMetaObject()
{
    if ( metaObj )
      return metaObj;
    QMetaObject* parentObject = QDialog::staticMetaObject();
#ifndef QT_NO_PROPERTIES
    static const QMetaEnum::Item enum_0[] = {
      { "NoIcon",  (int) QMessageBox::NoIcon },
      { "Information",  (int) QMessageBox::Information },
      { "Warning",  (int) QMessageBox::Warning },
      { "Critical",  (int) QMessageBox::Critical },
      { "Question",  (int) QMessageBox::Question }
    };
    static const QMetaEnum enum_tbl[] = {
      { "Icon", 5, enum_0, FALSE }
    };
#endif // QT_NO_PROPERTIES
    static const QUMethod slot_0 = {"buttonClicked", 0, 0 };
```

```
    static const QMetaData slot_tbl[] = {    //slot 元数据
      { "buttonClicked()", &slot_0, QMetaData::Private }
    };
#ifndef QT_NO_PROPERTIES  // 类声明中 Q_PROPERTY(...) 对应的项经 moc 转化后得到属性元数据
    static const QMetaProperty props_tbl[4] = {
      { "QString","text", 0x3000103, &QMessageBox::metaObj, 0, -1 },
      { "Icon","icon", 0x0107, &QMessageBox::metaObj, &enum_tbl[0], -1 },
      { "QPixmap","iconPixmap", 0x6000103, &QMessageBox::metaObj, 0, -1 },
      { "TextFormat","textFormat", 0x010f, &QMessageBox::metaObj, 0, -1 }
    };
#endif // QT_NO_PROPERTIES
//创建元对象
    metaObj = QMetaObject::new_metaobject(
      "QMessageBox", parentObject,
      slot_tbl, 1,
      0, 0,
#ifndef QT_NO_PROPERTIES
      props_tbl, 4,
      enum_tbl, 1,
#endif // QT_NO_PROPERTIES
      0, 0 );
    cleanUp_QMessageBox.setMetaObject( metaObj );
    return metaObj;
}
//判断类名是不是这个类
void* QMessageBox::qt_cast( const char* clname )
{
    if ( !qstrcmp( clname, "QMessageBox" ) )
      return this;
    return QDialog::qt_cast( clname );
}
//判断激活的是否是这个元对象槽函数
bool QMessageBox::qt_invoke( int _id, QUObject* _o )
{
    switch ( _id - staticMetaObject()->slotOffset() ) {
    case 0: buttonClicked(); break;
    default:
      return QDialog::qt_invoke( _id, _o );
    }
    return TRUE;
}
//判断发出信号的是否是这个元对象
bool QMessageBox::qt_emit( int _id, QUObject* _o )
{
    return QDialog::qt_emit(_id,_o);
}
#ifndef QT_NO_PROPERTIES
//判断是否是这个元对象的属性
bool QMessageBox::qt_property( int id, int f, QVariant* v)
{
    switch ( id - staticMetaObject()->propertyOffset() ) {
    case 0: switch( f ) {
      case 0: setText(v->asString()); break;
      case 1: *v = QVariant( this->text() ); break;
      case 3: case 4: case 5: break;
      default: return FALSE;
    } break;
    case 1: switch( f ) {
      case 0: setIcon((Icon&)v->asInt()); break;
      case 1: *v = QVariant( (int)this->icon() ); break;
      case 3: case 4: case 5: break;
      default: return FALSE;
    } break;
    case 2: switch( f ) {
      case 0: setIconPixmap(v->asPixmap()); break;
```

```
        case 1: if ( this->iconPixmap() ) *v = QVariant( *iconPixmap() ); break;
        case 3: case 4: case 5: break;
        default: return FALSE;
      } break;
      case 3: switch( f ) {
        case 0: setTextFormat((TextFormat&)v->asInt()); break;
        case 1: *v = QVariant( (int)this->textFormat() ); break;
        case 3: case 4: case 5: break;
        default: return FALSE;
      } break;
      default:
        return QDialog::qt_property( id, f, v );
      }
      return TRUE;
}

bool QMessageBox::qt_static_property( QObject* , int , int , QVariant* ){ return FALSE; }
#endif // QT_NO_PROPERTIES
```

$QTDIR/src/dialogs/qmessagebox.cpp 文件中的类 QMessageBoxLabel 声明含有 Q_OBJECT 宏，经 moc 编译器产生了 $QTDIR/src/.moc/release-shared/qmessagebox.moc 文件。QMessageBoxLabel 的声明后面加上了#include "qmessagebox.moc"语句， QMessageBoxLabel 声明列出如下：

```
class QMessageBoxLabel : public QLabel
{
    Q_OBJECT
public:
    QMessageBoxLabel( QWidget* parent ) : QLabel( parent, "messageBoxText")
    {
      setAlignment( AlignAuto|ExpandTabs );
    }
};
#include "qmessagebox.moc"  //包含有 moc 生成的元对象函数
```

2.6 进程间通信——QCopChannel

信号与槽提供了对象间通信的能力，而 QCopChannel 类提供了多个进程之间的通信能力。 QCOP（Communication Protocol）是 Qt 的一种通信协议，它允许不同的客户在同一进程或不同的进程之间进行通信。目前，这种 QCOP 机制仅用于 Qt/Embedded 中，在 Qt/X11 和 Windows 中使用 DCOP（Desktop Communication Protocol）和 COM 标准。

DCOP 的全称是“桌面通信协议”，是为 KDE 桌面系统开发的一种基于信号的 IPC/RPC机制。它是一个客户端到客户端的通信协议，以基于 X11R6 标准库 LibICE 的服务器为中间媒介。这个协议支持消息传送和使用 XML-RPC 到 DCOP“网关”的远端程序调用，并提供 KDE 应用程序的进程间及远程应用程序间的通信功能。

QCopChannel 继承自 QObject 类，它提供了 send()及 isRegistered()等静态函数。用户调用 QCopChannel 类构造函数创建一个信道，并重载 receive()函数或者使用 connect() 函数将信号与槽函数连接起来。receive()函数允许 QCopChannel 子类处理从其通道上接收的消息。

QCOP 机制使用了 Qt/Embedded 内部的客户/服务器通信机制， QCopChannel 类构造函数及 send 函数分别使用了如下代码：

```
// 通知服务器注册这个信道，qt_fbdpy 是 QDisplay 类的全局对象
qt_fbdpy->registerChannel( channel );

qt_fbdpy->sendMessage( channel, msg, data );
```

在嵌入设备的窗口系统 Qtopia（QPE）中提供了 QCopEnvelope 类用于进程间通信，该类实际上是对 QCopChannel 进行的封装，方便用户的使用。发送消息时使用 QCopEnvelope 类，接收消息使用 QCopChannel 类。

发送消息的方法如下：

```
QCopEnvelope e(channelname, messagename);
```

如果消息携带参数，则还需要将消息参数发送出去，方法如下：

```
e << parameter1 << parameter2 << ...;
```

Qtopia 中的通道名都以“ QPE/ ”开始，在接收消息时，通通道与 slot 函数相连接，方法如下：

```
myChannel = new QCopChannel( "QPE/FooBar", this );
connect( myChannel, SIGNAL(received(const QCString &, const QByteArray &)),
        this, SLOT(fooBarMessage( const QCString &, const QByteArray &)) );
```

示例：QCopChannel 通信过程

下面这个例子说明 QCopChannel 通信过程，首先创建一个通道，这个通道实际上是一个内存中的文件。然后在一个类中建立发送消息函数，在另一个类中建立接收消息函数，具体如下：

（1）在一个应用程序的类构造函数中定义通道；

```
//class1.cpp
MyClass1::MyClass1(QObject* parent, const char* name ) :
   QObject( parent, name )
{
   myChannel = new QCopChannel("System/Shell", this );
.....
}
```

（2）在一个应用程序的类中发送消息：

```
//class2.cpp
MyClass2::sentMessage(QObject* parent, const char* name ) :
   QObject( parent, name )
{
   ......
   QByteArray ba;
   QDataStream stream( ba, IO_WriteOnly );
   stream << QString("cat") << QString("file.txt");
   QCopChannel::send( "System/Shell", "execute(QString,QString)", ba );
}
```

（3）在另一个应用程序的类中接收消息：

```
void MyClass::receive( const QCString &msg, const QByteArray &data )
{
   QDataStream stream( data, IO_ReadOnly );
   if ( msg == "execute(QString,QString)" ) {
      QString cmd, arg;
      stream >> cmd >> arg;
      ...
   } else if ( msg == "delete(QString)" ) {
      QString filenname;
      stream >> filename;
      ...
   } else ...
}
```

2.7 窗口部件类

窗口部件的父类一般是 QWidget 类、QFrame类和 QScrollView 类，这 3个类是继承关系，分别是祖父、父、子的关系。QWidget 类是所有窗口部件的基类，QFrame 类是带有框架（如带有凸出边界的 QLabel 窗口部件）的窗口部件的基类。 QScrollView 类是视图窗口部件（如 QIconView 类）的基类。窗口部件类的继承关系见图 2-5。

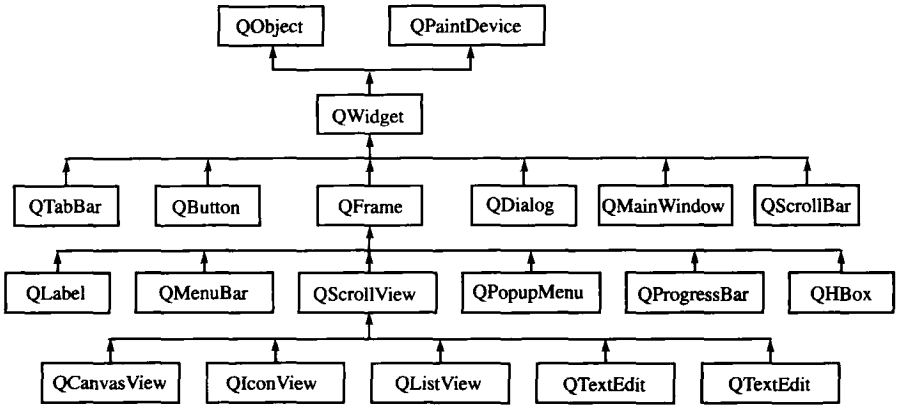


图 2-5 窗口部件类的继承关系

QWidget 类、QFrame 类和 QScrollView 类的窗口从外观上能看出区别，如图 2-6所示。

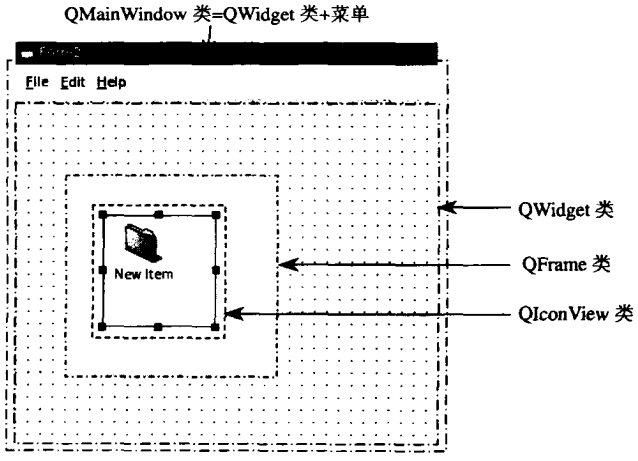


图 2-6 QWidget 类、QFrame 类和 QScrollView 类部件

下面说明这 3个窗口部件类的区别。

QWidget 类是所有窗口部件的基类，它在一个区域内定义了窗口部件的各种事件、窗口显示及窗口层次管理等基本功能函数，不能单独作为窗口部件使用。窗口部件类必须从 QWidget 类继承，如 QMainWindow类在继承 QWidget 类基础上添加了菜单栏。

QFrame 类从 QWidget 类继承，它含有框架，即用户指定的边框（如凸出的边框），并对边框的各种属性进行管理。因此，含有边框的窗口部件类都必须从其继承。

QScrollView 类从 QFrame 类继承，是各种视图窗口类的基类。当视图中的控件超出窗口显示范围时，视图会自动添加下滚动条。QScrollView 类在 QFrame 类的基础上，增加对视图特性的管理函数。

2.7.1 窗口部件的基类 QWidget

1. QWidget 类的基类说明

QWidget 类从 QObject 类和 QPaintDevice 类继承，QObject 类管理对象树、信号与槽机制及其元对象等；QPaintDevice 类是能被绘画的对象的基类，它是绘画设备的抽象，是一个能使用 QPainter 在其中绘制的二维空间抽象，被子类 Qwidget、Qpixmap、Qpicture 和 QPrinter 类继承。一个绘制设备的座标系统的原点在屏幕的左上角位置， X 轴向右增长，Y 轴向下增长，单位是像素点。

QPaintDevice 类使用 PDevCmd 枚举类型列举了常用的绘制命令，其成员函数可以设置或得到绘制设备的一些通用属性，如颜色映射和可视性等。对于 Qt/X11来说,调用 Xlib库函数 对于 Qt/Embedded 来说，使用 Qt/Embedded 服务器接口函数*qwsDisplay()等。全局的位图拷贝函数 bitBlt 提供了从源绘画设备对象向目的绘制设备对象进行像素拷贝。

2. QWidget 类功能说明

QWidget 类是所有用户界面对象的基类， QWidget 窗口部件是用户界面的基本单元。它是屏幕上的矩形区域窗口，多个窗口按 Z 轴顺序排列，从窗口系统接收鼠标、键盘和其他事件。一个窗口部件可以被其父窗口部件或者其前面的窗口部件盖住一部分。

未被嵌入到一个父窗口部件的窗口部件被叫做“顶级窗口部件”，一个没有父窗口部件的窗口部件一直是顶级窗口部件，通常顶级窗口部件是带有框架和标题栏的窗口。在 Qt 中，QMainWindow 和各种 QDialog 的派生类是最常见的顶级窗口。

非顶层窗口部件是子窗口部件，它们是其父窗口部件中的子窗口，通常不能可见地从其父窗口部件中区分一个子窗口部件。在 Qt 中的绝大多数其他窗口部件仅仅作为子窗口部件。

QWidget 有很多成员函数，但是其中的一些有几乎没有功能。有很多继承其子类提供的实际功能，比如 QPushButton、QListBox 和 QTabDialog 等。QWidget类成员函数分类如表 2-3 所示。

表 2-3 QWidget 类成员函数分类

函数分类	方法函数名
窗口函数	show()、hide()、raise()、lower()和 close()
顶级窗口	caption()、setCaption()、icon()、setIcon()、iconText()、setIconText()、isActiveWindow()、setActiveWindow()、showMinimized()、showMaximized()、showFullScreen()和 showNormal()
窗口内容	update()、repaint()、erase()、scroll()和 updateMask()
几何形状	pos()、size()、rect()、x()、y()、width()、height()、sizePolicy()、setSizePolicy()、sizeHint()、updateGeometry()、layout()、move()、resize()、setGeometry()、frameGeometry()、geometry()、childrenRect()、adjustSize()、mapFromGlobal()、mapFromParent()、mapToGlobal()、mapToParent()、maximumSize()、minimumSize()、sizeIncrement()、setMaximumSize()、setMinimumSize()、setSizeIncrement()、setBaseSize()和 setFixedSize()
模式	isVisible()、isVisibleTo()、visibleRect()、isMinimized()、isDesktop()、isEnabled()、isEnabledTo()、isModal()、isPopup()、isTopLevel()、setEnabled()、hasMouseTracking()、setMouseTracking()、isUpdatesEnabled()和 setUpdatesEnabled()
外观、风格	style()、setStyle()、cursor()、setCursor()、font()、setFont()、palette()、setPalette()、backgroundMode()、setBackgroundMode()、colorGroup()、fontMetrics()和 fontInfo()
键盘焦点函数	isFocusEnabled()、setFocusPolicy()、focusPolicy()、hasFocus()、setFocus()、clearFocus()、setTabOrder()和 setFocusProxy()
事件处理	event()、mousePressEvent()、mouseReleaseEvent()、mouseDoubleClickEvent()、mouseMoveEvent()、keyPressEvent()、keyReleaseEvent()、focusInEvent()、focusOutEvent()、wheelEvent()、enterEvent()、leaveEvent()、paintEvent()、moveEvent()、resizeEvent()、closeEvent()、dragEnterEvent()、dragMoveEvent()、dragLeaveEvent()、dropEvent()、childEvent()、showEvent()、hideEvent()和 customEvent()

续表

函数分类	方法函数名
变化处理	enabledChange()、fontChange()、paletteChange()、styleChange()和 windowActivationChange()
系统函数	parentWidget()、topLevelWidget()、reparent()、polish()、winId()、find()和 metric()
这是什么的帮助	customWhatsThis()
内部核心函数	focusNextPrevChild()、wmapper()、clearWFlags()、getWFlags()、setWFlags()和 testWFlags()

每一个窗口部件构造函数至少接受两个或 3 个标准参数，这些参数说明如下。

（1）QWidget *parent = 0：新窗口部件的父窗口部件。如果为 0 （默认），新的窗口部件将是一个顶级窗口部件；否则它将会是 parent 的一个孩子，并且被 parent 的几何形状所强迫（除非指定 WType_TopLevel 作为窗口部件标记）。

（2）const char *name = 0 ：新窗口部件的窗口部件名称，可以使用 name()来访问它。窗口部件名称很少被程序员用到，但是对于图形用户界面构造程序，如 Qt 设计器，是相当重要的（可以在 Qt 设计器中命名一个窗口部件，并且在代码中使用这个名字来连接它）。 dumpObjectTree() 调试函数也使用它。

（3）WFlags f = 0： （在可用情况下）设置窗口部件标记，默认设置对于几乎所有窗口部件都是适用的。但是举例来说，对于一个没有窗口系统框架的顶级窗口部件，则必须使用特定的标记。

2.7.2 QFrame 类

QFrame 类是有框架的窗口部件的基类，它绘制框架并可调用被子类重新实现的虚函数 drawContents()填充这个框架。还有另外两个较少用到函数 drawFrame()和 frameChanged()，它们被用来绘制框架。

例如，QPopupMenu 从 QFrame 类继承，可以将菜单边框凸出显示；QProgressBar 从 QFrame 类继承，可以将进度条有“凹陷”的外观。这些边框外观可以通过 QFrame 类方法函数来改变，方法列出如下：

```
QLabel label(...);
label.setFrameStyle( QFrame::Panel | QFrame::Raised );
label.setLineWidth( 2 );

QProgressBar pbar(...);
label.setFrameStyle( QFrame::NoFrame );
```

QFrame 类可以直接被用来创建没有任何内容的简单框架，但通常情况下使用 QFrame 类的子类 QHBox 或 QVBox，因为它们可以自动地布置放到 QFrame 窗口中的窗口部件。

QFrame 类窗口部件有 4 个属性描述窗口框架，即 frameStyle()、lineWidth()、midLineWidth()和 margin()。框架风格由框架外形和阴影风格决定，框架外形的种类有 NoFrame、Box、Panel、StyledPanel、PopupPanel、WinPanel、ToolBarPanel、MenuBarPanel、HLine 和 Vline；阴影风格的类型有 Plain、Raised 和 Sunken。

框架的外观还与线宽、中间线宽和边白有关，线宽指框架边界的宽度；中间线宽指在框架中间的另外一条线的宽度，它使用第 3 种颜色来得到一个三维的效果。注意中间线只有在 Box、HLine 和 VLine 这些凸起和凹陷的框架中才被绘制，边白指框架和框架内容之间的间隙。

由风格与线宽组成的框架外观组合如图 2-7 所示，其中列出了各种外观的 QFrame 类窗口部件。

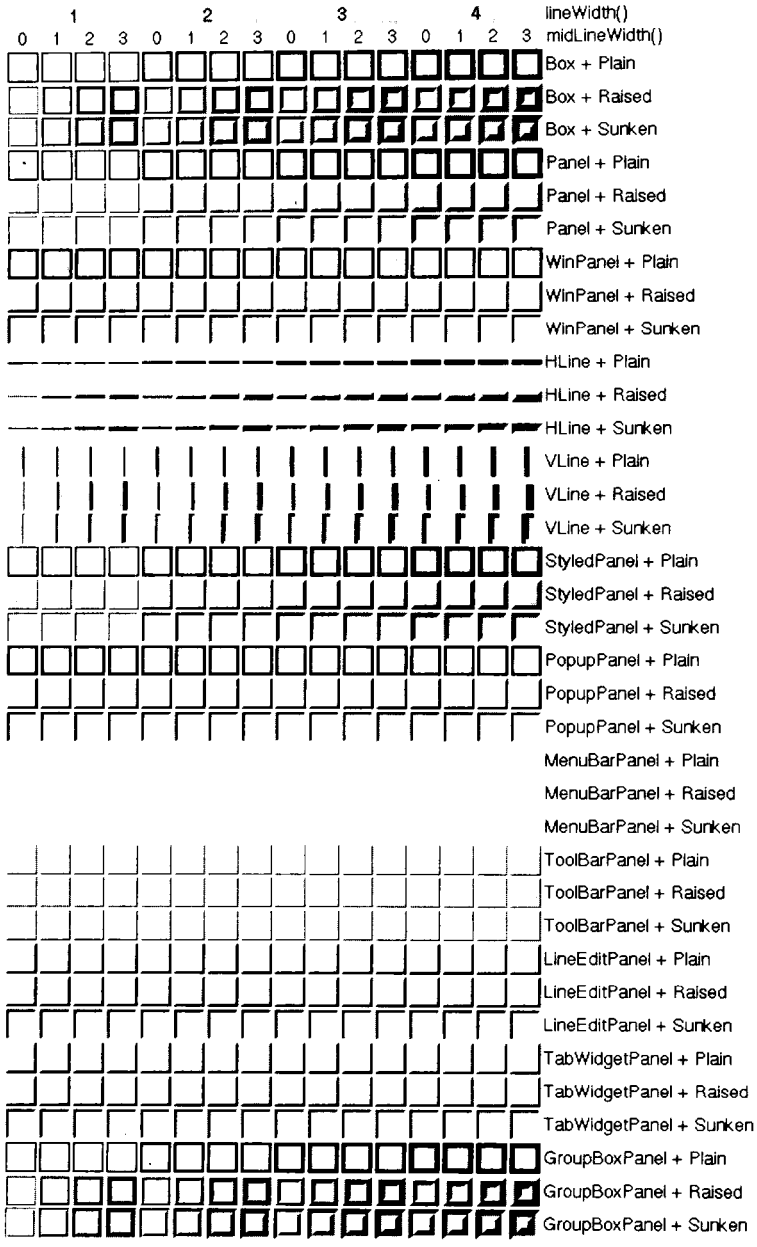


图 2-7 由风格与线宽组成的框架外观组合

2.7.3 QScrollView 类

QScrollView 窗口部件提供了在需要时可以出现滚动条的滚动区域，它是一个很大的画布，可比下面窗口支持的坐标大得多。例如，一个很大的网页，长度超过了窗口限制，必须使用滚动条拖动才能阅读超出窗口的内容。

QScrollView 类重新实现了 drawContents()函数，并且可以使用 resizeContents()设置查看区域的大小，使用 addChild()和 moveChild()在视图中增加和移动子窗口部件。

QScrollView 有 4个孩子窗口部件，分别是 viewport()、verticalScrollBar()、horizontalScrollBar() 和 cornerWidget()。viewport() 是视图在窗口中的可见部分，通过滚动条的滚动，从 viewport()上可按部分

依次查看整个视图的内容。

QScrollView 类有 3 种风格的窗口结构，即单个较大的子窗口部件、带有一些窗口部件的平面区域和带有许多窗口部件的平面区域。

（1）单个较大的子窗口部件

QScrollView 的最简单的用法是在一个不超过 4000 像素的滚动区域（这是 X11 服务器的最大可靠尺寸）中放入一个大的子窗口部件在 QScrollView 中。这个子窗口部件是滚动视图的 viewport()的孩子，它通过 addChild()被添加。在视图中添加一个子窗口部件的方法如下：

```
QScrollView* sv = new QScrollView(...); //创建视图
//给视图的 viewport()类添加单个子窗口部件
QVBox* big_box = new QVBox(sv->viewport());
sv->addChild(big_box); //为单个子窗口部件添加孩子
```

还可以继续在滚动视图中单一孩子中添加任意子窗口部件，方法如下：

```
QLabel* child1 = new QLabel("CHILD", big_box);
QLabel* child2 = new QLabel("CHILD", big_box);
QLabel* child3 = new QLabel("CHILD", big_box);
```

这样在视图中，viewport()有一个孩子——大的 QVBox，这个 QVBox 有 3 个 QLabel 对象作为子窗口部件。当视图被滚动时，QVBox 被移动。从 viewport()窗口可以依次按部分查看 QVBox 及其孩子，如同几个子窗口部件被移动了一样。 QScrollView 类的 viewport()子类有单个较大子窗口部件的情况如图 2-8 所示。

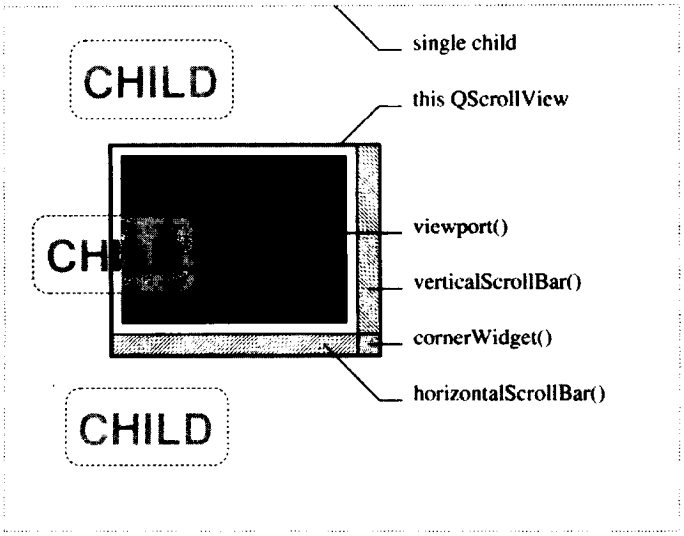


图 2-8 QScrollView 类的 viewport()子类有单个较大子窗口部件的情况

（2）带有一些窗口部件的平面区域

QScrollView 有非常大的滚动区域，在这个区域中少量窗口部件的高度或宽度超过了 4000 像素。这时需要调用 resizeContents()来设置区域的大小，并且重新实现 drawContents()来绘制内容。你可以使用 addChild()给 viewport()添加孩子，而不是给 viewport()的一个子类添加孩子。方法如下：

```
QScrollView* sv = new QScrollView(...);
QLabel* child1 = new QLabel("CHILD", sv->viewport());
sv->addChild(child1);
QLabel* child2 = new QLabel("CHILD", sv->viewport());
sv->addChild(child2);
```

```
QLabel* child3 = new QLabel("CHILD", sv->viewport());
sv->addChild(child3);
```

这样，viewport()有 3个 QLabel 对象作为子窗口部件。当视图被滚动， viewport() 会分别看到子窗口部件。QScrollView 带有一些窗口部件的平面区域的情况如图 2-9所示。

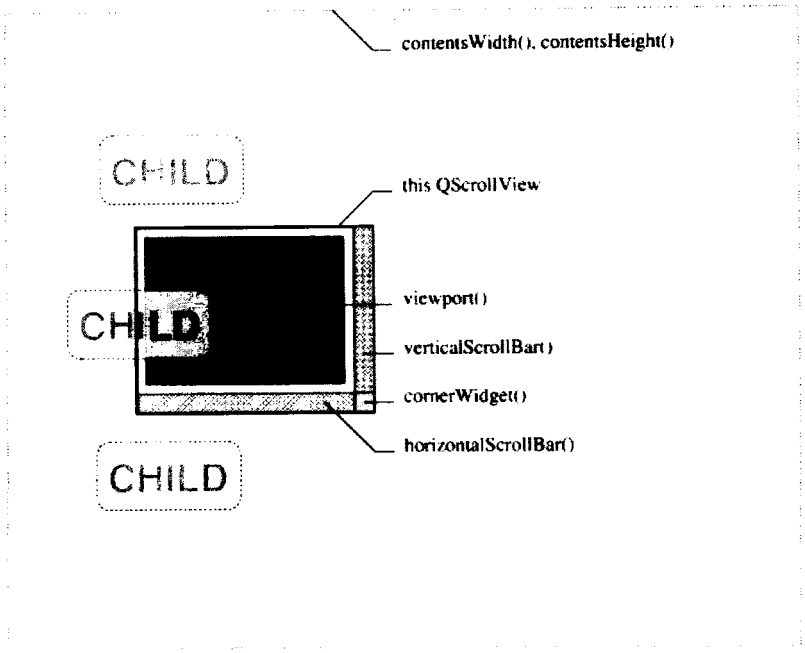


图 2-9 QScrollView带有一些窗口部件的平面区域的情况

（3） 带有多个窗口部件的平面区域

QScrollView 是非常大的滚动区域，在这个滚动区域中有多个窗口部件的高度或宽度超过 4000 像素。这时，需要调用 resizeContents()设置区域的大小并重新实现 drawContents() 来绘制内容。然后调用函数 enableClipper(TRUE) 把 viewport()设置为比窗口大的区域，并为 viewport()添加孩子。方法如下：

```
QScrollView* sv = new QScrollView(...);
sv->enableClipper(TRUE);
QLabel* child1 = new Qlabel("CHILD", sv->viewport());
sv->addChild(child1);
Qlabel* child2 = new Qlabel("CHILD", sv->viewport());
sv->addChild(child2);
Qlabel* child3 = new Qlabel("CHILD", sv->viewport());
sv->addChild(child3);
```

这样 clipper()有一个孩子 viewport()，viewport()有同样的 3 个标签作为子窗口部件。当视图被滚动时，viewport() 被移动，它的孩子如同通常的子窗口部件那样被移动。 QScrollView 带有多个窗口部件的平面区域的情况如图 2-10所示。

可以使用上面所述 3种方法中的一种来在视图中使用子窗口部件，在滚动区域中看到的窗口部件是 viewport()窗口部件，而不是 QScrollView 本身。因此只有使用 viewport()->setMouseTracking(TRUE)才能打开鼠标跟踪。

鼠标拖放事件被传送给父对象，为了使拖放生效，可以在 QScrollView 上设置 setAcceptDrops（TRUE）。对于视图来说，正常需要将鼠标在 viewport() 中的屏幕相对位置座标换算为视图的绝对座标，为此需要使用 viewportToContents()函数。

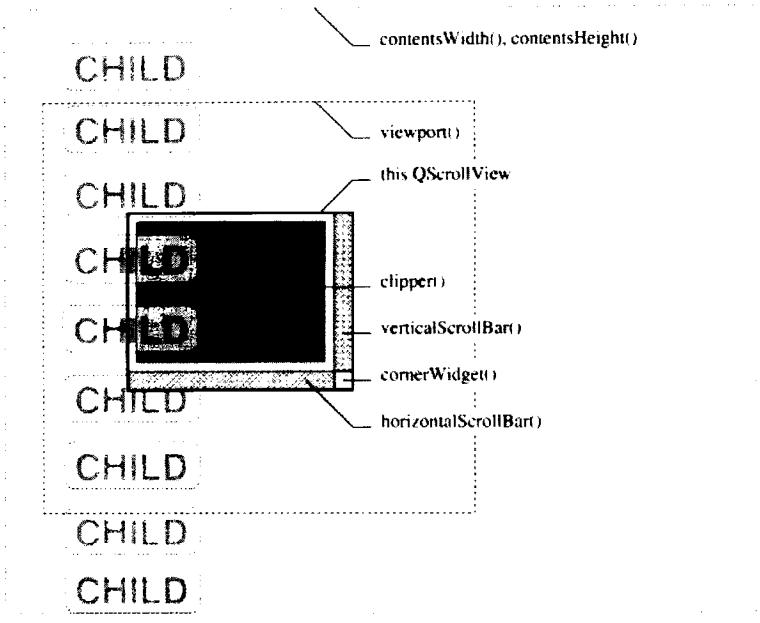


图 2-10　QScrollView 带有多个窗口部件的平面区域的情况

为了在滚动区域中处理鼠标事件，应重新实现 contentsMousePressEvent()，而不是 mousePressEvent()，因为 contentsMousePressEvent() 提供了滚动视图坐标系统中的转换事件。如果重新实现 mousePressEvent()，那么只有当 QScrollView 的 cornerWidget() 部件被点击时，mousePressEvent() 才能得到调用。

2.8　Qt 风格机制

Qt 的风格（style）机制实现了可以通过插件或风格类来改变的图形用户接口（GUI）的外观，例如 Unix 平台上通常使用 Motif 和 CDE 风格。在嵌入设备上常有主题（theme）或外观设置应用程序，可以为嵌入设备设置不同风格的 GUI 图形界面，这个应用程序使用的就是风格机制。

2.8.1　风格类

QStyle 是风格类的基类，为了设置窗口部件的 UI 界面风格，QStyle 类定义了大量的枚举类型和十几个函数。枚举类型表示界面上的不同元素（如组合框中的按钮及按钮的边框等），函数控制图形用户界面的绘制。QStyle 大多数成员函数只有声明而没有函数实现，其实现是通过在 QCommonStyle、QWindowStyle、QMotifStyle 及其子类中进行重载来完成的。

QStyle 类实现的 3 个函数函数说明如下。

（1）drawItem()：负责绘制文本和像素图。

（2）itemRect()：返回文本或图像所占的区域。

（3）visualRect()：返回逻辑坐标，这个函数使 Qt 实现 right-to-left 风格（阿文及维文传统是文本从右向左显示，因此控件布局也是从右向左）。

图 2-11 所示为 Qt 中与风格相关的类的继承关系。

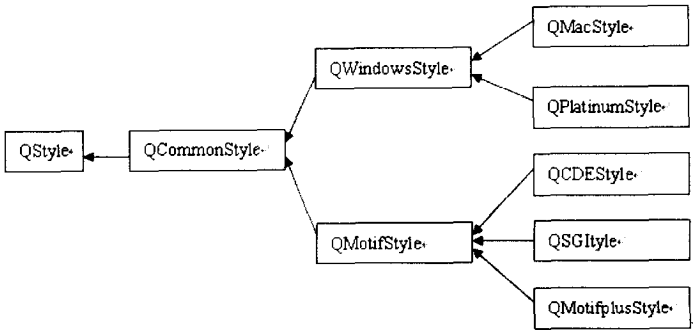


图 2-11 风格相关类的继承关系

按照对 GUI 图形界面的显示效果的影响，风格被分成不同枚举类型，每个枚举类型按照 Qt 的各种基本图形界面元素被分成不同的枚举项。表 2-4 所示为风格枚举类型的说明。

表 2-4 风格枚举类型的说明

风格枚举类型	风格枚举类型的说明
QStyle::ComplexControl	代表一个复杂控制类型（如 SpinWidget、ComboBox 和 TitleBar 等），它依赖于用户的点击或按键操作有不同的行为
QStyle::ContentsType	代表一个容积类型，它被用来计算各种 widgets 的容积尺寸
QStyle::ControlElement	代表一个控制元素，它是执行一些动作或显示信息给用户 widget 的一部分，如按钮的标签
QStyle::PixelMetric	代表一个像素点距离，它是风格依赖的尺寸，如 PM_ButtonMargin 表示按钮标签与框架之间的空白区域的大小
QStyle::PrimitiveElement	代表一个风格的基本图形界面元素，它是一个通常的 GUI 元素，如 QStyle::PE_ArrowUp 表示向上的箭头
QStyle::StyleFlags	代表一个基本图形界面元素的绘制标志，如 QStyle::Style_Enabled 表示 widget 被激活时的风格
QStyle::StyleHint	代表一个 StyleHint，它是一个常规的外观，如 QStyle::SH_ScrollBar_BackgroundMode 表示一个 QScrollBar 的背景模式
QStyle::StylePixmap	代表一个风格位图，它是遵循一些存在 GUI 风格的位图，如 QStyle::SP_MessageBoxWarning 表示警告信息的图标
QStyle::SubControl	代表在一个复杂控制的子控制
QStyle::SubRect	代表一个 widget 的子区域，Style 使用这些区域绘制 widget 的不同部分

如果用户需要修改界面风格的某一部分，就必须重载 QStyle 中描述这部分的相关绘制函数。下面简单地说明这几个绘制函数，详细的说明可参考 Qt 文档。

（1）drawPrimitive(.....)：在指定区域绘制基本图形元素，如 QSpinBox 中的带箭头的按钮等。

（2）drawComplexControl(.....)：在指定区域用指定颜色绘制复杂窗口部件，如 SpinWidget、comboBox、slider 和 listView 等。

（3）drawComplexControlMask(...)：在指定区域用指定的颜色以位掩码的方式绘制复杂窗口部件。

（4）drawItem (.....)：绘制指定区域中的文本或位图。

创建 Style 的方法是首先选择一个风格基类，在用户的风格类中重载基类的函数。然后在应用程序中使用这个新的风格类，或用一个插件来使用这个风格类。选择基类时，可以选择 QStyle 或其他风格派生类作为父类，如 QWindowsStyle 或 QMotifStyle。

示例：创建一个用户风格类 CustomStyle

在这个例子中，我们首先改变在 QWindowsStyle 风格下的标准箭头的外观。箭头是 Primitive Elements 类型（即 GUI 基本元素），使用函数 drawPrimitive() 绘制。在这个简单例子中，我们选择了 QWindowsStyle 作为父类。

CustomStyle 类声明列出如下：

```
#include <qwindowsstyle.h>

class CustomStyle : public QWindowsStyle {
   Q_OBJECT
public:
   CustomStyle();
   ~CustomStyle();

   void drawPrimitive( PrimitiveElement pe,  //绘制基本图形元素
                  Qpainter *p,
                  const Qrect & r, //表示一个矩形区域
                  /*表示一个部件(widget)的颜色组，color group 含有部件绘制自身时使用的各种颜色，如前景色和背景色等*/
                  const QcolorGroup & cg,
                  //控制如何绘制图形界面元素的标志
                  Sflags flags = Style_Default,
                  /*绘制不同的部件(widget)时会需要不同的参数，如绘制面板（panel）可能需要线宽作为额外参数*/
                  const QstyleOption & = QstyleOption::Default ) const;

private:
   //取消了拷贝构造函数和'='操作符。
   CustomStyle( const CustomStyle & );
   CustomStyle& operator=( const CustomStyle & );
};
```

下面是用户风格类 CustomStyle 的实现代码：

```
CustomStyle::CustomStyle()
{
}

CustomStyle::~CustomStyle()
{
}

void CustomStyle::drawPrimitive( PrimitiveElement pe,
                          QPainter * p,
                          const QRect & r,
                          const QColorGroup & cg,
                          SFlags flags,
                          const QStyleOption & opt ) const
{
   // 我们仅对箭头感兴趣
   if (pe >= PE_ArrowUp && pe <= PE_ArrowLeft) {
      QPointArray pa( 3 );
      // 使箭头覆盖其所支持的绘画区域的一半
      int x = r.x();
      int y = r.y();
      int w = r.width() / 2;
      int h = r.height() / 2;
      x += (r.width() - w) / 2;
      y += (r.height() - h) /2;

      switch( pe ) {
      case PE_ArrowDown:
```

```
            pa.setPoint( 0, x, y );
            pa.setPoint( 1, x + w, y );
            pa.setPoint( 2, x + w / 2, y + h );
            break;
        case PE_ArrowUp:
            pa.setPoint( 0, x, y + h );
            pa.setPoint( 1, x + w, y + h );
            pa.setPoint( 2, x + w / 2, y );
            break;
        case PE_ArrowLeft:
            pa.setPoint( 0, x + w, y );
            pa.setPoint( 1, x + w, y + h );
            pa.setPoint( 2, x, y + h / 2 );
            break;
        case PE_ArrowRight:
            pa.setPoint( 0, x, y );
            pa.setPoint( 1, x, y + h );
            pa.setPoint( 2, x + w, y + h / 2 );
            break;
        default: break;

        }

        //使用不同的颜色指示箭头 enabled/disabled 状态
        if ( flags & Style_Enabled ) {
            p->setPen( cg.mid() );
            p->setBrush( cg.brush( QColorGroup::ButtonText ) );
        } else {
            p->setPen( cg.buttonText() );
            p->setBrush( cg.brush( QColorGroup::Mid ) );
        }
        p->drawPolygon( pa );  //画箭头
    } else {
        // 让基类处理其他的基本元素
        QWindowsStyle::drawPrimitive( pe, p, r, cg, flags, data );
    }
}
```

在 Qt 应用程序中有多种方法使用用户风格，最简单的是在应用程序的 main()函数中添加下面的行：

```
int main( int argc, char ** argv )
{
    QApplication::setStyle( new CustomStyle() );
    //创建 QApplication 对象等其他通常的例程
......
}
```

注意还必须在项目文件中包括 customstyle.h 和 customstyle.cpp 文件。

还可以以插件的形式创建 style，这时 style 作为共享对象在运行时被装载。编译 style 插件，并把它放在$QTDIR/plugins/styles 目录中。对已存在的应用程序可以使用下面的命令在运行时自动装载 style 插件：

```
./application -style custom
```

2.8.2 窗口系统风格更新机制

Qt 本身包含了风格更新机制，函数 Global::applyStyle()是窗口系统风格更新的顶层函数。它通过系统信道“ QPE/System ”发送消息“ applyStyle()”到各个应用程序，应用程序会根据这个消息调用函

数 applyStyle 设置其窗口风格。函数 Global::applyStyle() 调用层次如图 2-12 所示。

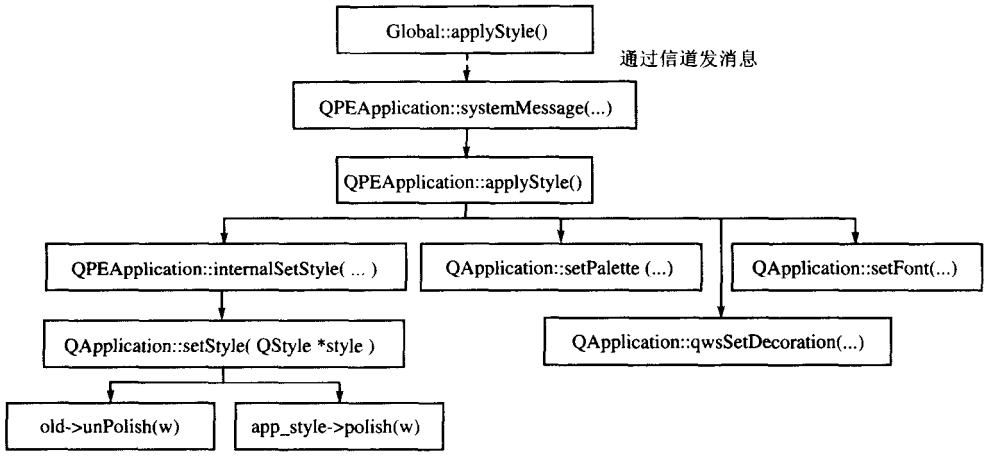


图 2-12 函数 Global::applyStyle()调用层次

函数 Global::applyStyle() 列出如下：

```
void Global::applyStyle()
{
#ifndef QT_NO_COP
    QCopChannel::send( "QPE/System", "applyStyle()" );
#else
    ((QPEApplication *)qApp)->applyStyle(); // apply without needing QCop for floppy version
#endif
}


void QPEApplication::systemMessage( const QCString &msg, const QByteArray &data)
{
    ......
    else if ( msg == "applyStyle()" ) {
      applyStyle();
    }
    ......
}
```

函数 applyStyle 应用风格，设置调色板及字体，并且设置应用程序的装饰风格类。该函数读取配置文件中的风格类名、字体名、颜色值及位图，为应用程序的各个窗口设置风格。方法是得到应用程序的窗口部件链表，遍历各个窗口部件。然后调用风格类函数 polish 函数装饰窗口，设置窗口部件的调色板颜色值及窗口部件的字体。

函数 applyStyle 分析如下：

```
void QPEApplication::applyStyle()
{
#ifdef Q_WS_QWS
    QString styleName;
    //读取 qpe.conf 配置文件中 Appearance 组中 Style 选项
    Config config( "qpe" );
    config.setGroup( "Appearance" );
    styleName = config.readEntry( "Style", "Qtopia" );

    //设置风格到窗口部件
    internalSetStyle( styleName );
```

```
    //从配置文件中读取颜色值，设置调色板
    setPaletteEntry( tempPal, config, Button, "#F0F0F0" );
    setPaletteEntry( tempPal, config, Background, "#EEEEEE" );
    ......
    setPalette( pal, TRUE );  //设置应用程序的默认调色板为pal

    color = config.readEntry( "AlternateBase", "#EEF5D6" );
    if ( color[0] == '#' ) //存在颜色值，使用颜色值
      style().setExtendedBrush(QStyle::AlternateBase, QColor(color));
    else {//不存在颜色值，则是位图名，装载位图
      QPixmap pix;
      pix = Resource::loadPixmap(color);
      style().setExtendedBrush(QStyle::AlternateBase, QBrush(QColor("#EEF5D6"),pix));
    }
   ......
    //读取窗口装饰配置，设置装饰对象，并重绘制窗口
    QString dec = config.readEntry( "Decoration", "Qtopia" );
    QString decTheme = config.readEntry( "DecorationTheme", "" );
    if ( dec != d->decorationName || !decTheme.isEmpty()) {
      qwsSetDecoration( new QPEDecoration( dec ) );// 设置装饰对象，并重绘制窗口
      d->decorationName = dec;
      d->decorationTheme = decTheme;
    }

    // 读取字体配置
    QString ff = config.readEntry( "FontFamily", font().family() );
    int fs = config.readNumEntry( "FontSize", font().pointSize() );
    QFont fn(ff,fs);

    //根据字体的大小算出图标高度，并将高度值存入配置文件
#ifndef QPE_FONT_HEIGHT_TO_ICONSIZE
#define QPE_FONT_HEIGHT_TO_ICONSIZE(x) (x+1)
#endif
    int is = config.readNumEntry( "IconSize", -1 );
    if ( is < 0 ) {
      QFontMetrics fm(fn);
      config.writeEntry( "IconSize", QPE_FONT_HEIGHT_TO_ICONSIZE(fm.height()) );
      config.write(); //写入图标高度（根据字体大小计算得到）
    }

    setFont( fn, TRUE ); //设置字体

#endif
}
```

函数 internalSetStyle 创建风格对象，并设置风格，列出如下：

```
void QPEApplication::internalSetStyle( const QString &style )
{
......
    QStyle *newStyle = 0;
    if ( style == "QPE" || style == "Qtopia" ) {
      newStyle = new QPEStyle;
    } else {
      newStyle = QStyleFactory::create(style);
    }

    if ( !newStyle ) {
      newStyle = new QPEStyle;
      d->styleName = "QPE";
    } else {
      d->styleName = style;
    }
```

```
    if (&qApp->style() != newStyle)
      setStyle( newStyle );  //设置风格

}
```

函数 setStyle 设置应用程序的 GUI 风格到参数 style，风格对象的拥有关系被传递给 Qapplication。这样在 QApplication 在删除或设置一个新风格时，将删除 style 对象。

在切换风格时，颜色调色板将被设置回初始或系统默认状态，这是必需的，因为某一风格必须采用完全与风格相适应的颜色调色板。

函数 setStyle 分析如下：

```
void QApplication::setStyle( QStyle *style )
{
    QStyle* old = app_style;
    app_style = style;
#ifdef Q_WS_X11
    qt_explicit_app_style = TRUE;
#endif // Q_WS_X11
//如果启动失败，即一个 QApplication 对象尚未创建，删除旧的风格对象
    if ( startingUp() ) {
      delete old;
      return;
    }

    // 清除旧的风格
    if (old) {
      if ( is_app_running && !is_app_closing ) {
          QWidgetIntDictIt it( *((QWidgetIntDict*)QWidget::mapper) );
          register QWidget *w;
          while ( (w=it.current()) ) {          //遍历所有的窗口部件
            ++it;
            if ( !w->testWFlags(WType_Desktop) &&  //除了 Qtopia desktop 外
                 w->testWState(WState_Polished) ) { // 已被润饰(Polished)
                old->unPolish(w);//在窗口部件动态润饰之前，必须取消前一次的装饰参数
            }
          }
      }
      old->unPolish( qApp );
    }

    /*在润饰应用程序之前关注某一 GUI 风格的可能的调色板需求，因为 style 可能通过函数 QApplication::setStyle()调用它自身
*/
    if ( !qt_std_pal )
      qt_create_std_palette(); //创建标准调色板对象 qt_std_pal
    QPalette tmpPal = *qt_std_pal;
    setPalette( tmpPal, TRUE );  //设置默认的应用程序调色板 app_pal = tmpPal

    // 使用新的风格初始化应用程序
    app_style->polish( qApp );

    //如果必要，重新润饰存在的窗口部件
    if (old) {
      if ( is_app_running && !is_app_closing ) {
          QWidgetIntDictIt it( *((QWidgetIntDict*)QWidget::mapper) );
          register QWidget *w;
          while ( (w=it.current()) ) {          //遍历所有的窗口部件
            ++it;
            if ( !w->testWFlags(WType_Desktop) ) { //除了 Qtopia desktop 外
              if ( w->testWState(WState_Polished) )
                app_style->polish(w);          //重新装饰
              w->styleChange( *old );
              if ( w->isVisible() ){
```

```
                w->update();
              }
            }
          }
        }
        delete old;
      }
}
```

窗口部件的装饰函数 polish 是用户的风格类中的重载函数，在这个函数中用户可以设置窗口部件的各种参数。每次窗口部件动态装饰之前，必须调用函数 unPolish 取消前一次的装饰参数。下面列出一个用户定义的 FreshStyle 风格的 polish 函数及 unpolish 函数：

```
void FreshStyle::polish( QWidget *w )
{
    if ( w->inherits( "QListBox" ) ||     //是从 QListBox 继承的窗口部件
      w->inherits( "QListView" ) ||
      w->inherits( "QPopupMenu" ) ||
      w->inherits( "QSpinBox" ) ) {
     QFrame *f = (QFrame *)w; //设置线条宽度
     f->setFrameShape( QFrame::StyledPanel ); //设置框架的样式
     f->setLineWidth( 1 );
    }
//设置窗口部件 w 窗口状态为 WState_GlobalBrushOrigin，即画刷所在窗口
    QStyleHackWidget::setGlobalBrushOrigin(w);
}

void FreshStyle::unPolish( QWidget *w )
{
    if ( w->inherits( "QListBox" ) ||
      w->inherits( "QListView" ) ||
      w->inherits( "QPopupMenu" ) ||
      w->inherits( "QSpinBox" ) ) {
     QFrame *f = (QFrame *)w;
     f->setFrameShape( QFrame::StyledPanel );  //设置框架的样式
     f->setLineWidth( 2 ); //设置线条宽度
    }

    QStyleHackWidget::clearGlobalBrushOrigin(w);
}
```

函数 qwsSetDecoration 设置 QWSDecoration 派生类对象并重新绘制窗口，这个方法仅在 Qt/Embedded 中使用，该函数分析如下：

```
void QApplication::qwsSetDecoration( QWSDecoration *d )
{
    if ( d ) {
      delete qws_decoration;
      qws_decoration = d; //得到 QWSDecoration 派生类对象，用于 QApplication 内部
      QWidgetList *widgets = topLevelWidgets(); //得到所有窗口部件的链表
      QWidgetListIt it( *widgets );
      QWidget *w;
      while ( (w=it.current()) != 0 ) {
         ++it;
         if ( w->isVisible() && w != desktop() ) {
           ((QETWidget *)w)->updateRegion();  //更新窗口各区域的大小
//发出重画事件重新绘制 desktop()->rect()区域
           ((QETWidget *)w)->repaintDecoration(desktop()->rect(), FALSE);
           if ( w->isMaximized() ) //如果已是最大化，则最大化显示
              w->showMaximized();
         }
      }
```

```
        delete widgets;
    }
}
```

2.9 布局类

Qt 的布局（layout）系统提供了设计子窗口部件布局的方法，其功能是在布局空间中布置子窗口部件（如水平、垂直或网格布置）、设置子窗口部件之间的空隙，以及管理在布局空间中布置子窗口部件等。

当添加一个窗口部件到一个布局中，布局中的所有窗口部件根据其 QWidget::sizePolicy()策略被分配到一定空间中并根据布局的需要来伸展窗口部件。

窗口部件基类 QWidget 中与布局相关的有属性 sizeHint 和尺寸策略类 QsizePolicy，分别说明如下。

（1）QSize sizeHint 属性：保存建议的窗口部件大小。

（2）QSizePolicy 类：描述了水平和垂直扩展大小的策略，一个窗口部件的尺寸策略是指窗口重定义大小的各种方式。QSizePolicy 使用枚举结构 SizeType 表示一个窗口部件尺寸策略，如 QSizePolicy::Fixed 表示窗口部件固定为 QWidget::sizeHint() 大小，QSizePolicy::Minimum 表示窗口部件最小尺寸为 QWidget::sizeHint()大小。QSizePolicy 使用枚举结构 ExpandData 表示一个窗口部件扩展策略，如 QSizePolicy::Horizontally 表示窗口部件能比 sizeHint() 大小扩展得更宽。

QLayout 是 Qt 的布局系统抽象基类，其中与布局相关的属性说明如下。

（1）int margin：布局的外边框的宽度。

（2）ResizeMode resizeMode：布局的扩展方式，如 QLayout::Minimum 表示主窗口部件的最小尺寸设置为 minimumSize() 不能再小。

（3）int spacing 布局内的窗口部件间的空白空间大小。

1. 布局窗口部件

让窗口部件有好布局的最好方法是使用布局窗口部件 QHBox、QVBox 和 QGrid，布局窗口类 QHBox、QVBox 和 QGrid 允许子窗口部件按水平、垂直和网格排列。为了生成更复杂的布局，可以嵌入一个布局窗口部件到其他布局窗口部件中。有 5 个 QLabel 子窗口部件，使用布局窗口部件 QHBox、QVBox 和 QGrid 后的显示如图 2-13 所示。

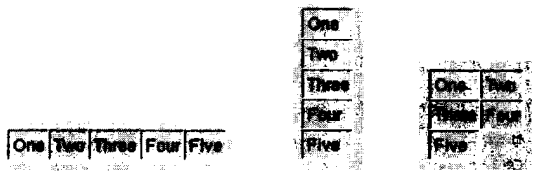


图 2-13 布局窗口类 QHBox、QVBox 和 QGrid 排列 QLabel 部件

图中所示的表格布局是由下面这些代码生成的：

```
QGrid *mainGrid = new QGrid( 2 ); // 一个 2*n 的网格
new QLabel( "One", mainGrid );
new QLabel( "Two", mainGrid );
new QLabel( "Three", mainGrid );
new QLabel( "Four", mainGrid );
new QLabel( "Five", mainGrid );
```

2. 布局过程

当添加窗口部件到一个布局中时，布局的工作过程如下。

（1）所有的窗口部件最初根据其 QWidget::sizePolicy()而被分配到一定空间中。

（2）如果任何一个窗口部件设置了大于 0 的伸展因子，则其就会按伸展因子比例分配空间。

（3）如果任何一个窗口部件的伸展因子为 0，那么只有当其他窗口部件不需要空间时才会得到更多的空间。其中，空间会首先根据放大策略被分配给窗口部件。

（4）分配了小于窗口部件的最小尺寸的任何窗口部件（或者是在没有定义最小尺寸时的最小默认尺寸）将使用窗口部件要求的最小尺寸。

（5）分配了大于窗口部件的最大尺寸的任何窗口部件，将使用窗口部件要求的最大尺寸。

放大因子被用来根据窗口部件互相间的比例来改变它们所被分配的空间，在没有设置放大因子时，QHBox 布置的 3 个窗口部件显示如图 2-14 所示。

如果我们为每个窗口部件设置一个放大因子，它们将被按比例布置（但不能小于最小默认尺寸），QHBox 布置的设置了放大因子 3 个窗口部件显示如图 2-15 所示。

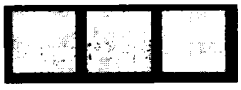


图 2-14　QHBox 布置的 3 个窗口部件



图 2-15　QHBox 布置的设置了放大因子 3 个窗口部件

3. 使用布局类

如果需要更多的控制布局，可以使用 QLayout 的子类。Qt 包括的布局类有 QGridLayout 和 QBoxLayout。QHBoxLayout 和 QVBoxLayout 是 QBoxLayout 的子类，它们使代码简单易读。

当使用一个布局时，则必须将布局子类插入到父窗口部件和布局类中。这样可以给出每个窗口部件的布局参数，定义诸如对齐方式、放大因子和位置这样的属性。

可以使用网格布局类 QGridLayout 将窗口部件按网格布置，网格布局类比网格窗口类 QGrid 可提供更多对窗口布局参数的控制。下面的代码生成网格布局：

```
QWidget *main = new QWidget;

// 生成一个 1*1 的网格布局对象，它可以自动自动展开
QGridLayout *grid = new QGridLayout( main, 1, 1 );

// 根据(行，列)编址来添加前 4 个窗口部件
grid->addWidget( new QLabel( "One", main ), 0, 0 );
grid->addWidget( new QLabel( "Two", main ), 0, 1 );
grid->addWidget( new QLabel( "Three", main ), 1, 0 );
grid->addWidget( new QLabel( "Four", main ), 1, 1 );

// 添加最后一个窗口部件到第 2 行，并占用第 0 列和第 1 列，中间对齐
grid->addMultiCellWidget( new QLabel( "Five", main ), 2, 2, 0, 1,
                          Qt::AlignCenter );

// 设置第 0 列和第 1 列的宽度比例为 2:3
grid->setColStretch( 0, 2 );
grid->setColStretch( 1, 3 );
```

可以插入一个子布局到父布局中，通过把父布局作为子布局构造函数的一个参数来把布局添加到另一个布局中。下面的代码是插入一个子布局到父布局的例子：

```
QWidget *main = new QWidget;
QLineEdit *field = new QLineEdit( main );
QPushButton *ok = new QPushButton( "OK", main );
QPushButton *cancel = new QPushButton( "Cancel", main );
QLabel *label = new QLabel( "Write once, compile everywhere.", main );
```

```
// 父窗口部件布局
QVBoxLayout *vbox = new QVBoxLayout( main );
vbox->addWidget( label );
vbox->addWidget( field );

// 在父布局中创建子布局
QHBoxLayout *buttons = new QHBoxLayout( vbox );
buttons->addWidget( ok );
buttons->addWidget( cancel );
```

如果对默认放置位置不满意，则创建没有父布局的布局对象，调用 addLayout()把这个布局插入到其他布局中。插入的布局对象成为子布局，被插入的布局对象变成父布局。

当创建自己的窗口部件类时，应该与其布局属性联系。如果这个窗口部件有一个 QLayout，则说明布局已被处理；如果这个窗口部件没有任何子窗口部件，或者没有人工定义布局，则应该重实现 QWidget 的下面成员函数。

（1）QWidget::sizeHint() ：返回窗口部件的优先选用的大小。

（2）QWidget::minimumSizeHint()：返回窗口部件所能有的最小尺寸。

（3）QWidget::sizePolicy()：返回一个 QsizePolicy，其值描述了这个窗口部件所需要的空间。

当默认尺寸、最小默认尺寸和尺寸策略发生改变时，应调用函数 QWidget::updateGeometry()，从而引起布局的重新计算。对 updateGeometry() 的多次调用只会引起一次重新计算。

如果窗口部件优先选用的高度依赖于其实际宽度（如一个自动断词的标签），在 sizePolicy() 中设置 hasHeightForWidth() 标识，并且重新实现函数 QWidget::heightForWidth()。

即使实现了 heightForWidth()，还应提供一个好的函数 sizeHint()。sizeHint()提供了窗口部件优先选用宽度，并且被不支持 heightForWidth() 的 QLayout 子类使用（QGridLayout 和 QBoxLayout 都支持 heightForWidth()）。

还可以通过 QWidget::event()来进行人工布局。当布局需要重新计算的时候，窗口部件会得到 LayoutHint 类型事件，重新实现函数 QWidget::event() 并处理 LayoutHint 类型事件。

4. 设计布局类

设计一个用户自己的布局类需要以 QLayout 为基类，首先创建一个 QptrList 对象用来存放布局处理的条目对象。每个条目是一个 QLayoutItem，它常对应一个窗口部件，接着实现一个 QGLayoutIterator 类的派生类来实现对 QPtrList 对象（链表）中每个条目的查找。然后才能实现对每个条目的布局的管理，如为每一个子窗口部件调用 setGeometry()来设置尺寸。

如果要生成一种特殊的布局，则重新实现 QWidget::resizeEvent() 来计算所需要分配的大小并且为每一个子窗口部件调用 setGeometry()。

当布局需要重新计算的时候，窗口部件会得到一个类型是 LayoutHint 的事件，重新实现被通知 LayoutHint 事件窗口部件的 QWidget::event()。

下面分别说明 QLayoutItem 和 QGLayoutIterator 的作用。

QLayoutItem 类提供了 QLayout 操作的抽象条目，如布局空间的一个窗口部件对象，这个类被用户的布局类使用。它提供了纯函数返回布局的信息，包括 sizeHint()、minimumSize()、maximumSize() 和 expanding()。通过 setGeometry()和 geometry()函数可设置或得到布局的几何尺寸，通过 iterator()返回布局的条目，如具体的窗口部件。

QGLayoutIterator 是内部布局迭代器的抽象基类，用于得到用户布局的各个条目，必须实现的函数是 next()、current()和 takeCurrent()。

示例：用户的布局类 CardLayout

下面是布局类 CardLayout 的声明，其中使用一个 QPtrList 存放被布局处理的条目，每一个条目是

一个 QLayoutItem 对象：

```
card.h

#ifndef CARD_H
#define CARD_H

#include <qlayout.h>
#include <qptrlist.h>

class CardLayout : public QLayout
{
public:
    CardLayout( QWidget *parent, int dist )
        : QLayout( parent, 0, dist ) { }
    CardLayout( QLayout* parent, int dist)
        : QLayout( parent, dist ) { }
    CardLayout( int dist )
        : QLayout( dist ) { }
    ~CardLayout();

    void addItem(QLayoutItem *item); //如何在布局中添加一个 item
    QSize sizeHint() const;    // 布局优先选用的尺寸
    QSize minimumSize() const; // 最小尺寸
    QLayoutIterator iterator(); // 用于遍历查找布局中的每个条目
    void setGeometry(const QRect &rect);

private:
    QPtrList<QLayoutItem> list; //    存放在布局中的各个对象（如：窗口部件）
};

#endif
```

CardLayoutIterator 类用于遍历查找 QPtrList 链表中存放的对象，实现 QGLayoutIterator 类中的虚函数 current()、next()和 takeCurrent()。

CardLayoutIterator 类的声明和实现列出如下：

```
card.cpp

#include "card.h"
class CardLayoutIterator : public QGLayoutIterator
{
public:
    CardLayoutIterator( QPtrList<QLayoutItem> *l )
        : idx( 0 ), list( l ) { }

    QLayoutItem *current()  //从对象链表中得到当前项
    { return idx < int(list->count()) ? list->at(idx) : 0; }

    QLayoutItem *next()    //从对象链表中得到下一项
    { idx++; return current(); }

    QLayoutItem *takeCurrent()
    { return list->take( idx ); } //从对象链表中删除当前项

private:
    int idx;
    QPtrList<QLayoutItem> *list;
};
```

在 CardLayout 类中创建 iterator()，使用类 CardLayoutIterator 来构造：

```
QLayoutIterator CardLayout::iterator()
{
   return QLayoutIterator( new CardLayoutIterator(&list) );
}


void CardLayout::addItem( QLayoutItem *item )  //在布局中加入条目
{
   list.append( item );
}


CardLayout::~CardLayout()
{
   deleteAllItems();  //从布局中删除所有条目
}
```

下面是布局的实现函数：

```
 void CardLayout::setGeometry( const QRect &rect )  //设置尺寸
 {
    QLayout::setGeometry( rect );  //设置布局的区域大小

    QPtrListIterator<QLayoutItem> it( list );
    if (it.count() == 0)
       return;

    QLayoutItem *o;

    int i = 0;
 //使用 spacing()得到条目（窗口部件）之间的空白距离
    int w = rect.width() - ( list.count() - 1 ) * spacing();
    int h = rect.height() - ( list.count() - 1 ) * spacing();

    while ( (o = it.current()) != 0 ) {
       ++it;
       QRect geom( rect.x() + i * spacing(), rect.y() + i * spacing(),
                w, h );
       o->setGeometry( geom );
       ++i;
    }
 }

 QSize CardLayout::sizeHint() const
 {
    QSize s( 0, 0 );
    int n = list.count();
    if ( n > 0 )
       s = QSize( 100, 70 ); // start with a nice default size
    QPtrListIterator<QLayoutItem> it( list );
    QLayoutItem *o;
    while ( (o = it.current()) != 0 ) {
       ++it;
       s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
 }

 QSize CardLayout::minimumSize() const
 {
    QSize s( 0, 0 );
    int n = list.count();
    QPtrListIterator<QLayoutItem> it( list );
    QLayoutItem *o;
```

```
    while ( (o = it.current()) != 0 ) {
        ++it;
        s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
}
```

2.10 Qt 插件

Qt 提供了一个简单的插件接口，可以轻松地生成作为独立组件的定制数据库驱动、图像格式、文本编解码器（text codec）、风格（style）和独立的窗口部件的控件。写一个插件需要选择合适的插件基类，有 5 个插件基类。从其派生的插件默认保存于标准插件目录下，见表 2-5。

表 2-5 保存插件基类及插件的默认路径

基　类	默认路径
QImageFormatPlugin	$QTDIR/plugins/imageformats
QSqlDriverPlugin	$QTDIR/plugins/sqldrivers
QStylePlugin	$QTDIR/plugins/styles
QTextCodecPlugin	$QTDIR/plugins/codecs
QWidgetPlugin	$QTDIR/plugins/designer

每种基类的插件必须要实现哪些虚拟函数根据基类而定，可以在一个插件中实现任意数目 的插件子类。只要它们都是由同一个基类，如所有插件子类都从 QStylePlugin 派生出来的。

示例：用户风格类插件

下面这个例子说明如何创建一个用户风格类 MyStyle 的插件 MyStylePlugin：

```
class MyStylePlugin : public QStylePlugin
{
public:
    MyStylePlugin() {}
    ~MyStylePlugin() {}
     //返回在插件中实现的类的一个字符串列表
    QStringList keys() const {
        return QStringList() << "MyStyle";
    }
    //返回所要的类的一个对象
    QStyle* create( const QString& key ) {
        if ( key == "MyStyle" )
            return new MyStyle;
        return 0;
    }
};

Q_EXPORT_PLUGIN( MyStylePlugin )
```

对于数据库驱动、图像格式、定制部件和文本编解码器，不需要显式的对象生成过程，Qt 会按需求查找并生成它们。风格则是例外，因为开发者可能想以代码显式地设置一个风格。应用一个风格，可以使用与下面类似的代码：

```
QApplication::setStyle( QStyleFactory::create( "MyStyle" ) );
```

如果插件保存在的插件的默认目录中，Qt 的应用程序会自动识别哪些插件可用，应用程序中不需

要任何查找和装载插件的代码。插件可以与应用程序一起编译，或者插件编译为动态链接库（在 Linux 中以 .so 结尾的文件，在 Windows 中以 .dll 结尾的文件）。

2.11 Qt 模板库

Qt 模板库（QT Template Library，QTL）是一套提供对象容器的模板。如果编译器没有适当的 STL（标准模板库)可用，QTL 将被代替使用。QTL 提供了对象的链表、对象的矢量（动态数组）、从一个类型到另个类型的映射（或称为“字典”）和相关的迭代器和算法。一个容器是包含和管理其他对象的个对象，并且提供迭代器对被包含的对象进行访问。 Qt 模板类说明如表 2-6 所示。

表 2-6 Qt 模板类说明

模板类名称	说 明
QMap	提供基于值的一个字典的模板类
QMapConstIterator	QMap 的常量迭代器
QMapIterator	QMap 的迭代器
QPair	提供基于值的一对元素的模板类
QValueList	提供基于值的一个双向链表的模板类
QValueListConstIterator	QValueList 的常量迭代器
QValueListIterator	QValueList 的迭代器
QValueStack	提供基于值的一个堆栈模板类
QValueVector	提供基于值的一个动态数组模板类

QTL 类的命名约定与其他 Qt 类一致（如 count()和 isEmpty()），并且提供额外的函数来兼容 STL 算法。如 size()和 empty()，可以像使用 STL 的函数 map 一样来使用它们。

与 STL 相比，QTL 仅仅包含了 STL 容器应用程序接口的最重要的特性。没有平台差异，通常要慢一些并且经常扩展为更少的对象代码。

如果不想拷贝存储对象，则最好使用 QPtrCollection 及派生类，它们被设计用来处理各种类指针。QObject 没有拷贝构造函数，因此不能作为一个值使用，但可以存储指向 QObject 的指针到 QValueList。当然，直接使用 QPtrList 更好。QptrList 如同所有其他基于 QPtrCollection 的容器一样，提供了比速度优化且基于值的容器更多健全的检查。

如果有一些使用值的对象，并且在目标平台中没有可用的 STL，Qt 模板库即可替代它，使用值的对象至少需要一个拷贝构造函数、一个赋值操作符和一个默认构造函数（如一个没有任何参数的构造函数）。

注意个快速的拷贝构造函数对于容器的高性能是关键的，因为许多拷贝操作将会发生。如果需要排序数据，则必须在数据类中实现 operator<()。

Qt 模板库是为高性能而设计，迭代器（Iterators）是非常快的。为了实现这样的性能，Qt 模板库比基于 QPtrCollection 的集合类执行更少的错误检查。一个 QTL 容器，例如 QTL 容器没有跟踪任何相关的迭代器。这样在诸如删除条目时没有执行有效性检查，但却提供了很好的执行性能。

2.11.1 迭代器

Qt 模板库打交道的是值对象，而不是指针对象。迭代器是最好的遍历容器方法，遍历一个容器可使用下面的循环：

```
typedef QValueList<int> List;
List l;
for( List::Iterator it = l.begin(); it != l.end(); ++it )
    printf( "Number is %i\n", *it );
```

begin()返回第 1 个元素的迭代器，end()返回的最后一个元素之后的一个迭代器。 end()标明的是一个无效的位置，它永远不能被解除引用。它只是任何一次迭代的终止条件，迭代可以从 begin()或 end()开始。同样的概念也适用于其他容器类，例如用于 QMap 和 QValueVector 的迭代方法如下：

```
typedef QMap<QString,QString> Map;
Map map;
for( Map::iterator it = map.begin(); it != map.end(); ++it )
   printf( "Key=%s Data=%s\n", it.key().ascii(), it.data().ascii() );

typedef QValueVector<int> Vector;
Vector vec;
for( Vector::iterator it = vec.begin(); it != vec.end(); ++it )
   printf( "Data=%d\n", *it );
```

2.11.2 算法

Qt 模板库定义了大量操作容器的算法，这些算法用模板库函数实现，并且提供了有迭代器的容器的通用代码。例如：qHeapSort()和 qBubbleSort()提供了著名的堆排序和冒泡排序算法，可以如下使用它们：

```
typedef QValueList<int> List;
List l;
l << 42 << 100 << 1234 << 12 << 8;
qHeapSort( l );

List l2;
l2 << 42 << 100 << 1234 << 12 << 8;
List::Iterator b = l2.find( 100 );
List::Iterator e = l2.find( 8 );
qHeapSort( b, e );

double arr[] = { 3.2, 5.6, 8.9 };
qHeapSort( arr, arr + 3 );
```

第 1 个例子对整个列表排序；第 2 个例子对两个迭代器之间的所有元素排序，即 100、1234 和 12；第 3 个例子表明迭代器是作为指针使用的。

一些常用的模板函数说明如下。

（1）函数 qSwap()用来交换两个变量的值，例如：

```
QString second( "Einstein" );
QString name( "Albert" );
qSwap( second, name );
```

（2）函数 qCount()用于统计容器中一个值出现的次数，例如：

```
QValueList<int> l;
l.push_back( 1 );       //放入 1 到 l 链表中
l.push_back( 1 );
l.push_back( 1 );
l.push_back( 2 );
int c = 0;
qCount( l.begin(), l.end(), 1, c ); //统计 1 的个数 c，  c = 3
```

（3）函数 qFind()用于查找容器中一个值的第一次出现位置，例如：

```
QValueList<int> l;
l.push_back( 1 );
l.push_back( 1 );
l.push_back( 1 );
l.push_back( 2 );
//查找 2 所在的位置
QValueListIterator<int> it = qFind( l.begin(), l.end(), 2 );
```

（4）函数 qFill()用于将一个值拷贝填充到一个范围，例如：

```
 QValueVector<int> v(3);
qFill( v.begin(), v.end(), 99 ); //将 99 填充整个 v 数组， v 包含 99, 99, 99
```

（5）函数 qEqual()用来比较两个范围的元素是否相等，两个范围的元素个数不一定相等。只要第 1 个范围的元素与第 2 个范围的对应元素都相等，就认为这两个范围相等，例如：

```
QValueVector<int> v1(3);
v1[0] = 1;
v1[2] = 2;
v1[3] = 3;

QValueVector<int> v2(5);
v1[0] = 1;
v1[2] = 2;
v1[3] = 3;
v1[4] = 4;
v1[5] = 5;

bool b = qEqual( v1.begin(), v2.end(), v2.begin() );
// b == TRUE
```

（6）函数 qCopy()用于拷贝一个范围的元素到输出迭代器，例如：

```
QValueList<int> l;
l.push_back( 100 );
l.push_back( 200 );
l.push_back( 300 );
QTextOStream str( stdout );
//拷贝 l 中所有元素到输出迭代器 QTextOStreamIterator
qCopy( l.begin(), l.end(), QTextOStreamIterator(str) );
```

（7）函数 qCopyBackward()用于拷贝一个容器或其一部分到一个输出迭代器，拷贝的次序是从后面开始，例如：

```
QValueVector<int> vec(3);
vec.push_back( 100 );
vec.push_back( 200 );
vec.push_back( 300 );
QValueVector<int> another;
// “another”包含的是按倒序排列的（300、200、100）。
qCopyBackward( vec.begin(), vec.end(), another.begin() );
```

如果编写了新的算法，则考虑将其写成模板函数，这样即可使它们能够用在尽可能多的容器上。在上一个例子中，可以很容易地使用 qCopy()打印出一个标准 C++数组，方法列出如下：

```
int arr[] = { 100, 200, 300 };
QTextOStream str( stdout );
qCopy( arr, arr + 3, QTextOStreamIterator( str ) );
```

2.11.3 数据流串行化

所有提到的容器（如 QValueList 、QStringList、QValueStack 和 QMap 等）都可被相应的流操作符串行化，下面是一个例子：

```
QDataStream str(...);
QValueList<QRect> l;
// ……在这里填充这个列表
str << l;
```

容器还能如下所示被再一次读入：

```
QValueList<QRect> l;
str >> l;
```

2.12 集合类

一个集合类是装有多个条目的容器，每个条目是某种数据结构，集合类能执行对容器中的条目的插入、删除及查找等操作。

Qt 有多个基于值和基于指针的集合类，基于指针的集合类使用指向条目的指针来工作；而基于值的集合类存储其条目的拷贝，类似于 STL 容器类，能和 STL 算法和容器一起使用。

基于值的集合类说明如表 2-7所示。

表 2-7 基于值的集合类说明

类名称	说 明
QValueList	基于值的链表
QValueVector	基于值的矢量结构
QValueStack	基于值的栈结构
QMap	基于值的字典结构

基于指 针的集合类说明如表 2-8所示。

表 2-8 基于指针的集合类说明

类名称	说 明
QCache 和 QIntCache	LRU（least recently used)缓存结构
QDict、QIntDict 和 QPtrDict	字典结构
QPtrList	双向链接的链表结构
QPtrQueue	FIFO 先进先出(first in,first out)队列结构
QPtrStack	LIFO 后进先出(last in, first out)栈结构
QPtrVector	矢量结构

QMemArray 是一个例外，它既不基于指针，也不基于值，而是基于内存的结构。在有简单数据结构的数组中使用 QMemArray 效率最高，它在拷贝和数组元素比较时使用位逻辑运算符操作。

这些类中有些具有迭代器，迭代器是遍历集合类中的条目的类。在 Qt 模板库中，基于值的集合和算法集成在起。本节讨论基于指针的容器。

2.12.1 基于指针的容器的结构

基于指针的容器有 4 个内部基类(QGCache、QGDict、QGList 和 QGVector) 操作 void 类型指针。通过增加/删除条目指针，一个由这 4 个类组成的薄模板层实现了实际的集合。

允许 Qt 的模板类的策略使得在空间上很经济（实现这些模板类仅增加了对基类的内联调用），而且还不影响执行效率。

示例：使用 QPtrList

下面的例子说明了如何在一个链表中存储 Employee 条目，并将其以相反的次序打印出来：

```
#include <qptrlist.h>
#include <qstring.h>
#include <stdio.h>

class Employee
{
public:
   Employee( const char *name, int salary ) { n=name; s=salary;
   const char *name()   const            { return n; }
   int        salary() const            { return s; }
private:
   QString    n;
   int        s;
};

int main()
{
   QPtrList<Employee> list;       // 指向 Employee 的指针链表
   list.setAutoDelete( TRUE );    //当链表条目被移动时，删除条目

   list.append( new Employee("Bill", 50000) );  //链表追加新的对象
   list.append( new Employee("Steve",80000) );
   list.append( new Employee("Ron",  60000) );

   QPtrListIterator<Employee> it(list); //遍历 Employee 链表
   for ( it.toLast(); it.current(); --it) ) {  //从尾向头遍历
      Employee *emp = it.current();
      printf( "%s earns %d\n", emp->name(), emp->salary() );
   }

   return 0;
}
```

程序运行结果如下：

```
Ron earns 60000
Steve earns 80000
Bill earns 50000
```

2.12.2 管理集合条目

所有基于指针的集合继承了 QPtrCollection 基类，这个类仅知道集合中的条目个数和删除策略。

当集合中的条目被移去时，默认时它们不被删除， QPtrCollection::setAutoDelete() 定义了删除策略。在上述 QPtrList 使用示例中，我们激活了自动删除功能来进行链表删除。

当插入一个条目到一个集合时，仅指针被拷贝，而不是拷贝条目本身，这称为“浅拷贝”。当插入一个条目时，拷贝所有条目的数组到集合中也是可能的，这称为“深拷贝”。

所有的集合类函数在插入条目时调用虚拟函数 QPtrCollection::newItem()，如果想进行深拷贝，则需要重载它。

当从一个链表中移去一个条目时，调用虚拟函数 QPtrCollection::deleteItem()。如果自动删除功能被激活，在所有集合类中的默认实现函数被调用来删除条目。

基于指针的集合类，如 QPtrList<type> 定义了指向对象的指针集合。我们在这里只讨论 QPtrList 类，其他基于指针的集合类和所有集合类迭代器都有同样的使用方法。

模板实例化方法如下：

```
QPtrList<Employee> list;
```

在这个例子中，条目的类或类型是 Employee，它必须在链表定义之前被定义。例如：

```
class Employee {
   ...
};
QPtrList<Employee> list;
```

2.12.3 迭代器

QPtrListIterator 能在链表被修改的同时非常安全地遍历链表，在同一个集合中，多个迭代器能独立地工作。

QPtrList 有一个指向所有迭代器的内部链表，这些迭代器当前操作链表。当一个链表条目被移去时，链表更新所有指向这个条目的迭代器。

QDict 和 QCache 集合没有遍历函数，为了遍历集合，必须使用 QDictIterator 或 QCacheIterator。

Qt 预定义的集合类有字符串链表 QstrList，QstrIList（在 qstrlist.h 中）和 QstringList（在 qstringlist.h 中）。在绝大多数情况下，应选择 QStringList，它是一个共享的 QString Unicode 字符串的值链表。QPtrStrList 和 QPtrStrIList 仅存储字符指针，而不是字符串本身。

基于指针的集合类和相关的迭代器类说明如表 2-9 所示。

表 2-9 基于指针的集合类和相关的迭代器类说明

类 名 称	说 明
QAsciiCache	基于 char* keys 的缓存的模板类
QAsciiCacheIterator	QAsciiCache 集合的迭代器
QAsciiDict	基于 char* keys 的字典的模板类
QAsciiDictIterator	QAsciiDict 集合的迭代器
QBitArray	位数组
QBitVal	与 QBitArray 一起用的内部类
QBuffer	在 QByteArray 上操作的 I/O 设备
QByteArray	字节数组
QCache	基于 QString keys 的模板类
QCacheIterator	QCache 集合的迭代器
QCString	C 语言中 0 结尾的字符数组(char *)
QDict	基于 QString keys 的字典的模板类
QDictIterator	QDict collections 的迭代器
QIntCache	基于 long keys 的缓存的模板类

续表

类 名 称	说　明
QIntCacheIterator	QIntCache 集合的迭代器
QIntDict	基于 long keys 的字典的模板类
QIntDictIterator	QIntDict 集合的迭代器
QObjectList	QObjects 的 QPtrList 链表
QObjectListIt	QObjectLists 的迭代器
QPtrCollection	大多数基于指针的 Qt 集合的基类
QPtrDict	基于 void* keys 的字典的模板类
QPtrDictIterator	QPtrDict 的迭代器
QPtrList	双向链表的模板类
QPtrListIterator	QPtrList 集合的迭代器
QPtrQueue	一个队列的模板类
QStrIList	对于大小写敏感的 char*双向链接的链表
QStrList	char* 双向链接的链表

2.13 Qt 线程

Qt 对线程提供了支持，它引入了一些基本与平台无关的线程类、线程安全传递事件的方式和全局 Qt 库互斥量，允许从不同线程调用 Qt 的方法。Qt 中与线程应用相关的类如表 2-10 所示。

表 2-10 Qt 中与线程应用相关的类

类 名 称	说　明
QMutex	线程之间的互斥锁
QSemaphore	整数信号量
QThread	与平台无关的线程
QWaitCondition	线程之间允许等待/唤醒的条件

使用线程需要 Qt 提供相应的线程库的支持，因此在编译安装 Qt 时，需要加上线程支持选项。

当在 Windows 操作系统上编译 Qt 时，线程支持是在一些编译器上的一个选项。在 Windows 操作系统上编译应用程序时，通过在 qconfig.h 文件中增加一个选项来解决这个问题。

在 Mac OS X 和 Unix 上编译 Qt 时，应在运行 configure 脚本时添加 -thread 选项。在 Unix 平台上，多线程程序必须用特殊的线程支持库连接。多线程程序必须连接线程支持库 libqt-mt，而不是标准的 Qt 库。编译应用程序时，应该使用宏定义 QT_THREAD_SUPPORT 来编译（如编译时使用 -DQT_THREAD SUPPORT ）。

2.13.1 线程类 QThread

在 Qt 中提供了 QThread 线程类，它提供了创建一个新线程的方法。线程通过重载 QThread::run() 函数开始执行，这一点与 Java 中的线程类相似。

示例：一个简单的线程

下面的例子实现了一个简单的继承自 QThread 的用户线程类，并运行两个线程。线程 b 在线程 a 运行后运行，代码如下：

```
class MyThread : public QThread {
public:
   virtual void run();
};

void MyThread::run()   //运行线程
{
   for( int count = 0; count < 20; count++ ) {
      sleep( 1 );
      qDebug( "Ping!" );
   }
}

int main()
{
   MyThread a;
   MyThread b;
   a.start();  //通过调用run()函数来执行
   b.start();
   a.wait();
   b.wait();
}
```

只有一个线程类是不够的，对于支持多线程的程序来说，还需要保护两个不同的线程对数据的同时访问，因此 Qt提供了 QMutex 类。一个线程可以锁住互斥量，当互斥量被锁住时，将阻塞其他线程访问临界数据，直到这个线程释放互斥量。这样，可以保护临界数据一次只能被一个线程访问。

Qt 库互斥量(qApp->lock() 和 qApp->unlock())是在访问 Qt 的 GUI 界面资源时用到的互斥量，在 Qt 中没有使用互斥量而调用一个函数通常会导致不可预知的行为。从另外一个线程中调用 Qt 的一个 GUI 相关函数需要使用 Qt 库互斥量。在这种情况下，所有访问图形或窗口系统资源的函数都与 GUI 相关。如果对象仅被一个线程访问，使用容器类、字符串或者输入/输出类不需要任何互斥量。

2.13.2 线程安全的事件传递

在 Qt 中，一个线程总是一个事件线程，线程从窗口系统中拉出事件并且把它们分发给窗口部件。静态方法 QThread::postEvent 从线程，而不是从事件线程中邮递事件。事件线程被唤醒并且事件如同一个正常窗口系统的事件一样在事件线程中被分发，例如，可以从不同的线程强制一个窗口部件进行重绘，方法如下：

```
QWidget *mywidget;
QThread::postEvent( mywidget, new QPaintEvent( QRect(0, 0, 100, 100) ) );
```

上述代码将异步地使 mywidget 在其区域中重绘一块 100×100 的正方形区域。

另外，还需要一些机制使得处于等待状态的线程在给定条件下被唤醒， QWaitCondition 类提供了这种功能。线程等待的条件 QWaitCondition 满足，QWaitCondition 表明发生的事件，将阻塞直到这件事件发生。当发生给定的事件时，QWaitCondition 将唤醒等待该事件的所有线程或者唤醒任意一个被选中的线程。它和 POSIX 线程条件变量具有相同功能，是 Unix 中的一种实现。

示例：应用 QWaitCondition 类

下面这个例子的功能是按下按钮后唤醒 worker线程，这个线程在按钮上显示工作状态，即 Waiting (等待)和 Working （正在工作）。当按钮被按下时，worker 线程正在工作，那么对线程不产生影响。当 run 函数再次循环到 mycond.wait()时，线程阻塞并等待。当按钮再次被按下时，触发 slotClicked() 函数运行，唤醒等待的线程。

```
#include <qapplication.h>
```

```
#include <qpushbutton.h>

// 全局条件变量
QWaitCondition mycond;

// Worker 类实现
class Worker : public QPushButton, public QThread
{
   Q_OBJECT

public:
   Worker(QWidget *parent = 0, const char *name = 0)
         : QPushButton(parent, name)
   {
      setText("Start Working");

      // 将 QPushButton 继承来的信号与槽 slotClicked()连接起来。
      connect(this, SIGNAL(clicked()), SLOT(slotClicked()));

      // 调用从 QThread 继承来的 start()方法开始线程的执行
      QThread::start();
   }

public slots:
   void slotClicked()
   {
      // 唤醒等待这个条件变量的一个线程
      mycond.wakeOne();
   }

protected:
   void run() //重载 run 函数
   {
        while ( TRUE ) {
          // 锁定应用程序互斥锁，并且设置窗口标题来表明我们正在等待开始工作
          qApp->lock();
          setCaption( "Waiting" );
          qApp->unlock();

          // 等待直到被告知可以继续
          mycond.wait();

          // 如果到了这里，表示已经被另一个线程唤醒
          qApp->lock();
          setCaption( "Working!" );// 设置标题，表示正在工作
          qApp->unlock();
      }
   }
};

int main( int argc, char **argv )
{
   QApplication app( argc, argv );

   // 创建一个 worker
   Worker firstworker( 0, "worker" );

   app.setMainWidget( &worker ); //将 worker 设置为应用程序的主窗口。
   worker.show();

   return app.exec();
}
```

当进行线程编程时，要注意的事项如下。

（1）在持有 Qt 库互斥量时不要执行任何阻塞操作，否则将冻结事件循环。

（2）确认锁定一个递归 QMutex 的次数等于解锁的次数，不能多也不能少。

（3）在调用除了 Qt 容器和工具类外的任何内容之前锁定 Qt 应用程序互斥量。

（4）谨防隐含的共享类，如果需要在线程之间指定它们，则应该用 detach() 分离它们。

（5）小心没有被设计成线程安全的 Qt 类，例如，QPtrList 的 API 接口不是线程安全的。并且如果不同的线程需要遍历一个 QPtrList ，它们应该在调用 QPtrList::first()之前锁住，在到达终点后解锁。

（6）确认仅在 GUI 线程中创建继承自 QWidget、QTimer 和 QSocketNotifier 的对象，在一些平台上创建在线程 中，而不是 GUI 线程的对象永远不会接收到底层窗口系统的事件。

（7）和上 面很相似，只在 GUI 线程中使用 QNetwork 类。因为所有的 QNetwork 类都是异步的，没有必要把 QSocket 用在多线程中。

（8）永远不要尝试在不是 GUI 线程的线程中调用 processEvents() 函数，也包括 QDialog::exec()、QPopupMenu::exec()、QApplication::processEvents() 和其他一些函数。

（9）在应用程序中，不要把普通的 Qt 库和支持线程的 Qt 库混合使用。这意味着如果应用程序使用了支持线程的 Qt 库，则不能连接普通的 Qt 库、动态地载入普通 Qt 库或者连接其他依赖普通 Qt 库的库或者插件。在一些系统中，这样做会导致 Qt 库中使用的静态数据崩溃。

2.14 鼠标拖放

拖放提供了一种用户在应用程序之间或之内传递信息的一种简单可视机制，在术语中，被称为“直接操作模型”。拖放在功能上类似剪贴板的剪切和粘贴机制，拖放机制包括拖动、放下 、剪贴板、拖放操作、添加新的拖放类型、高级拖放，以及和其他应用程序之间的操作几个方面。

1. 拖动

开始一个拖动，如鼠标移动事件。创建一个适合媒体的 QDragObject 的子类的对象，例如对于文本使用 QTextDrag，对于图片使用 QImageDrag 。然后调用 drag()方法，例如，从一个窗口部件中开始拖动一些文本：

```
void MyWidget::startDrag()
{
  QDragObject *d = new QTextDrag( myHighlightedText(), this )
  d->dragCopy();  //拷贝选中文本
  // 不要删除 d。
}
```

注意在拖动之后，QDragObject 没有被删除。在拖放明显完成后，这个 QDragObject 需要被保存。因为它还可能需要与其他进程通信，最后 Qt 会删除这个对象。如果拥有拖动对象的窗口部件在删除拖动对象之前被删除，那么任何没有完成的放下操作将会被取消，并且拖动对象会被删除。因为这个原因，可以应该小心对待对象引用。

2. 放下

为了能在一个窗口部件中接收被放下的媒体，这个窗口部件调用 setAcceptDrops（TRUE）（如在其构造函数中），并且重载事件处理方法 dragEnterEvent() 和 dropEvent()。对于更复杂的应用程序，重载 dragMoveEvent()和 dragLeaveEvent()也是必需的。

例如，当拖动后放下文本或图片时，窗口部件接受并处理放下操作的代码如下：

```
MyWidget::MyWidget(...) :
  QWidget(...)
```

```
{
   ...
   setAcceptDrops(TRUE); //接收被放下的媒体
}
//当一个拖动正在进行并且鼠标进入这个窗口部件，这个事件处理函数被调用
void MyWidget::dragEnterEvent(QDragEnterEvent* event)
{
   event->accept( QTextDrag::canDecode(event) ||
      QImageDrag::canDecode(event) );
}
//当拖动在这个窗口部件上被放下，这个事件处理器被调用。
void MyWidget::dropEvent(QDropEvent* event)
{
   QImage image;
   QString text;

   if ( QImageDrag::decode(event, image) ) {//解码图片
      insertImageAt(image, event->pos()); //在窗口部件中插入图片
   } else if ( QTextDrag::decode(event, text) ) {
      insertTextAt(text, event->pos());
   }
}
```

3. 剪贴板

QDragObject、QDragEnterEvent、QDragMoveEvent和 QDropEvent 类都是 QMimeSource（提供类型信息的类）的子类。如果在 QDragObject 中基于数据传递，则不仅可使用拖放，而且还可以使用传统的剪切和粘贴。QClipboard 有如下两个函数：

```
setData(QMimeSource*)
QMimeSource* data()const
```

使用这些函数，可以把拖放初始信息放到剪贴板中：

```
void MyWidget::copy()
{
   QApplication::clipboard()->setData( new QTextDrag(myHighlightedText()) );
}

void MyWidget::paste()
{
   QString text;
   if ( QTextDrag::decode(QApplication::clipboard()->data(), text) )
      insertText( text );
}
```

甚至能使用 QDragObject 的子类作为文件 I/O 部分，例如，如果应用程序有一个 QDragObject 的子类把 CAD 设计编码成 DXF 格式，则可以如下所示存储和装载这个格式的文件：

```
void MyWidget::save()
{
   QFile out(current_file_name);
   out.open(IO_WriteOnly);
   MyCadDrag tmp(current_design); // MyCadDrag 是 QDragObject 的子类
   out.writeBlock( tmp->encodedData( "image/x-dxf" ) );
}

void MyWidget::load()
{
   QFile in(current_file_name);
   in.open(IO_ReadOnly);
   if ( !MyCadDrag::decode(in.readAll(), current_design) ) {
```

```
            QMessageBox::warning( this, "Format error",
                tr("The file \"%1\" is not in any supported format")
                .arg(current_file_name)
            );
        }
    }
```

4. 拖放操作

在一些简单的情况下，拖放的目标接收一个被拖动数据的拷贝并且由源来决定是否删除初始的拖动对象，这是 QDropEvent 中的“ Copy”操作。目标也可以选择理解其他操作，特别是“ Move”和“Link”操作。如果目标理解了“ Move ”操作，则目标负责拷贝和删除操作，源不会尝试删除数据；如果目标理解为“ Link ”操作，则存储自身的引用到初始信息中，并且源不会删除初始信息。最通用的拖放操作是在同一个窗口部件中执行一个“ Move ”操作。

拖动操作的另一个主要用途是当使用一个引用类型，如 text/uri-list 时，实际上被拖动的数据是文件或对象的引用。

5. 添加新的拖放类型

拖放不仅仅局限于文本和图片，任何信息都可以被拖放。为了在应用程序之间拖放信息，两个应用程序必须指明彼此都能接收和产生的数据格式，可以通过使用 MIME 类型来获得。拖动的源提供一个它能产生的 MIME类型列表（按从最合适到最少合适的顺序排列），并且放下的目标选择一种它能接受的类型。例如，QTextDrag 提供了“text/plain” MIME 类型(普通没有格式的文本),还有“text/utf16”和“ text/utf8”的Unicode 格式的类型。QImageDrag提供了“ image/* ”类型，*是 QImageIO 支持的任何一种图片格式。并且 QUriDrag 子类提供了“text/uri-list ”的支持，它是传输一个文件名列表（或 URL）的标准格式。

为了实现一些还没有可用 QDragObject 子类的信息类型的拖放，首先和最重要的步骤是查找合适的存在格式，IANA（Internet Assigned Numbers Authority ）在 ISI（Information Sciences Institute） 中提供了一个 MIME 媒体类型的分级列表，使用标准的 MIME 类型将会使应用程序现在及未来能更好地与其他软件互相操作。

为了支持另外的媒体类型，从 QDragObject 或 QStoredDrag 派生类。当需要提供多种媒体类型的支持时，从 QDragObject 派生类。当一个类型足够时，从更简单的 QStoredDrag 派生类。

QDragObject 的子类将会重载 const char* format(int i) const 和 QByteArray encodedData(const char* mimetype) const 成员，并且提供一套方法编码媒体数据，提供静态成员 canDecode()和 decode()解码输入的数据。QImageDrag 的成员函数 bool canDecode (QMimeSource*) const 和 QByteArray decode (QMimeSource*) const 在子类中需要类似的重载。

QStoredDrag 的子类提供了提供一套方法编码媒体数据，静态成员 canDecode()和 decode()对进入的数据进行解码。

6. 高级拖放

在剪贴板模式中，用户可以剪切或复制资源信息，然后粘贴它。相似地，在拖放模式中，用户可以拖动信息的拷贝或者拖动信息本身到一个新的位置（移动）。拖放模式对于程序员来说都是更复杂的，程序直到放下（粘贴）完成才会知道用户是想剪切还是复制。在应用程序之间拖动没有什么区别，但是在一个应用程序之内进行拖动，应用程序必须小心不要将拷贝粘贴到同一个地方。例如，在同上窗口部件中拖动文本，拖动的开始点和放下事件处理函数应如下这样重载：

```
void MyEditor::startDrag()
{
    QDragObject *d = new QTextDrag(myHighlightedText(), this);
    if ( d->drag() && d->target() != this)
```

```
        cutMyHighlightedText(); //剪切选中的文本
}

void MyEditor::dropEvent(QDropEvent* event)
{
    QString text;

    if ( QTextDrag::decode(event, text) ) {
        if ( event->source() == this && event->action() == QDropEvent::Move ) {
            // 在同一个窗口部件时，不能使用粘贴拷贝，而应是移到到这个位置
            event->acceptAction();
            moveMyHighlightedTextTo(event->pos());
        } else {
            pasteTextAt(text, event->pos()); //粘贴拷贝
        }
    }
}
```

一些窗口部件在数据被拖动到它们上面时需要指定“是”或“否”接收，例如，一个 CAD 程序也许只接收在视图中的文本对象上放下的文本。在这种情况下， dragMoveEvent()被使用并且给定接受或者忽略拖动的区域。代码如下：

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
{
    if ( QTextDrag::canDecode(event) ) {
      MyCadItem* item = findMyItemAt(event->pos());
      if ( item )
        event->accept();
    }
}
```

7. 和其他应用程序之间的操作

在 X11 上，拖动使用公有的 XDND 协议。而 Qt在 Windows 上使用 OLE 标准，Qt 在 Mac 上使用 Carbon 拖动管理器。在 X11 上，XDND 使用 MIME，所以不需要转换，Qt 的应用编程接口与平台无关；在 Windows 上，识别 MIME 的应用程序可以通过使用 MIME 类型的剪贴板格式名字进行通信。一些 Windows 应用程序已经对其剪贴板格式使用 MIME 命名规范了。在内部，Qt 有能力在专有的剪贴板格式和 MIME 类型之间转换。在 X11 上，Qt 也支持使用 Motif 拖放协议的拖动。

2.15 键盘焦点

Qt 的窗口部件在图形用户界面中按用户的习惯方式来处理键盘焦点，基本出发点是用户的击键能定向到屏幕上窗口中的任何一个和窗口中的任何一个部件中。当用户按下一个键，他们期望键盘焦点能够到达正确的位置，并且软件必须尽量满足这种希望。系统必须确定击键定位在哪一个应用程序、应用程序中的哪一个窗口和窗口中的哪一个窗口部件。

2.15.1 焦点移动的方式

把焦点定位特殊的窗口部件的习惯方式如下。

（1）用户按下 Tab 键（或者 Shift+Tab 键，或者 Enter 键）。

（2）用户点击一个窗口部件。

（3）用户按下一个键盘快捷键。

（4）用户使用鼠标滚轮。

（5）用户移动焦点到一个窗口，并且应用程序必须决定窗口中的哪个窗口部件应该得到焦点。这些移动机制的每一个都是不同的，并且不同类型的窗口部件只能接收其中一些方式的焦点。

1．Tab 键或者 Shift+Tab 键

按 Tab 键是到目前为止用键盘移动焦点的最通用的方法，有时在输入数据的应用程序中 Enter 键和 Tab 键的作用是一样的，我们暂时忽略这一点。

所有窗口系统中使用有关焦点的最通用的方法是按 Tab 键移动键盘焦点到每个窗口的窗口部件循环列表中的下一个窗口部件。Tab 键按照循环列表的一个方向移动焦点， Shift+Tab 键按另一个方向移动焦点。按 Tab 键从一个窗口部件到下一个窗口部件移动焦点的次序叫做“ Tab 键次序 ”。

在 Qt 中，窗口部件循环列表存放在 QFocusData 类中。每个窗口有一个 QFocusData 对象，并且当选择合适的 QWidget::FocusPolicy 焦点策略的 QWidget::setFocusPolicy()被调用的时候，窗口部件自动将其追加到列表的末尾。可以使用 QWidget::setTabOrder()来自定义 Tab 键控制次序，如果没有定义这个次序，那么 Tab 键会按照窗口部件构造的顺序移动焦点。 Qt designer 工具提供了一个可视化的改变 Tab 键控制次序的方法。

因为按 Tab 键是如此常用，所以大多数含有焦点的窗口部件应该支持 Tab 焦点。主要例外情况是几乎未用到的窗口部件，并且在窗口部件上有一些移动焦点的键盘快捷键或者错误处理。

2．用户点击一个窗口部件

在使用鼠标或者其他指针设备的计算机中，用鼠标点击一个窗口部件是一种比按 Tab 键更为常用的方法。

当鼠标点击把焦点移到一个窗口部件时，对于编辑器窗口部件，它也会移动文本光标（窗口部件的内部焦点）到鼠标被点击的地点。

鼠标点击移动焦点是大多数窗口部件必须支持的，有时窗口部件需要避免鼠标点击移动焦点。例如，在一个字处理程序中，当用户点击 “B” （粗体）工具按钮，键盘焦点应该保留在原来的位置。在 Qt 中，只有 QWidget::setFocusPolicy() 函数影响点击焦点。

3．用户按下一个键盘快捷键

使用键盘快捷键来移动焦点不是很常用，这种情况可能会隐含地发生在打开的模式对话框中。但是也会显式地发生在使用焦点加速器中，例如在 QLabel::setBuddy()、QGroupBox 和 QTabBar 提供的加速器中。

用户想让焦点跳到的窗口部件都应支持快捷键焦点，例如一个 Tab 对话框为其每一个页提供键盘快捷键，所以用户可以按下诸如 Alt+P 键来跳到打印页面。但只能有少量的快捷键，并且为命令提供键盘快捷键也很重要，如在标准快捷键列表中， Alt+P 也可以用来粘贴、播放或打印。

4．用户使用鼠标滚轮

在 Microsoft Windows 中，鼠标滚轮的用法是一直由有键盘焦点的窗口部件处理。在 Mac OS X 和 X11 中，则由获得其他鼠标事件的窗口部件处理。

Qt 处理这种平台差异的方法是当使用滚轮时，让窗口部件移动键盘焦点。每个窗口部件上有合适的焦点策略，应用程序可以在 Windows、Mac OS X 和 X11 上按照习惯正确地处理焦点。

5．用户移动焦点到某个窗口

在这种情况下，应用程序必须决定窗口中的哪一个窗口部件接收焦点， Qt 自动实现这样的做法。即如果焦点以前在这个窗口中，那么窗口中有焦点的最后一个窗口部件应该重新获得焦点；如果以前焦点就从来没有来到过这个窗口，并且程序知道焦点应该从哪里开始，则在调用 QWidget::show() 显示

它之前，在应该接收焦点的窗口部件上调用 QWidget::setFocus() 。如果程序不知道，Qt 会选择一个合适的窗口部件。

2.15.2 焦点策略及操作函数

键盘焦点的策略及操作函数说明如下。

（1）焦点策略属性变量

focusPolicy 焦点策略属性变量保存窗口部件接收键盘焦点的策略。如果窗口部件通过 Tab 键来接收键盘焦点，这个策略就是 QWidget::TabFocus ；如果窗口部件通过点击来接收键盘焦点，这个策略就是 QWidget::ClickFocus；如果窗口部件使用上述两种方式，则是 QWidget::StrongFocus 。如果它不接收焦点（QWidget 的默认值），则是 QWidget::NoFocus。

如果一个窗口部件处理键盘事件，则必须使键盘焦点生效，通常在窗口部件的构造函数中完成。例如，QLineEdit 的构造函数调用 setFocusPolicy(QWidget::StrongFocus)。

（2）void QWidget::setFocus()[虚槽]

函数 setFocus()把键盘输入焦点赋给这个窗口部件（或其焦点代理）。

首先，一个焦点移出事件会被发送给焦点窗口部件（如果有的话）通知它关于失去焦点的事件。然后一个焦点进入事件被发送给这个窗口部件通知它刚刚接收到焦点。如果焦点移出和进入的窗口部件是同一个的话，则没有发生任何事件。

函数 setFocus()会把焦点给一个窗口部件，而忽略其焦点策略，但是不会清空任何键盘捕获（grabKeyboard()）。注意如果窗口部件是被隐藏的，则其不接收焦点。

（3）void QWidget::setFocusProxy(QWidget * w)[虚]

函数 setFocusProxy 设置这个窗口部件的焦点代理为窗口部件 w。如果 w 为 0，这个函数重置这个窗口部件没有焦点代理。

一些窗口部件，如 QComboBox 能够“拥有焦点”，但创建一个子窗口部件来实际处理这个焦点。例如，QComboBox 创建了一个 QLineEdit来处理焦点。

当“这个窗口部件”获得焦点时， setFocusProxy()设置的这个窗口部件实际获得焦点。如果有了一个焦点代理，focusPolicy()、setFocusPolicy()、setFocus()和 hasFocus()都在这个焦点代理上操作。

2.16 会话管理

一个会话是一组正在运行的应用程序，其中每一个都有一个特殊的状态。会话被一个称为“会话管理器”（Session manager）的服务程序来控制，在会话中每个参与的应用程序被称为“会话客户”。

会话管理器为用户发出命令给它的客户，这些命令会使客户提交没有保存的变化（如保存打开的文件），使客户为将来的会话保存状态或关机。这样的一些操作被称为“会话管理”。

通常情况下，一个会话由用户在其桌面上同时运行的所有应用程序组成。在 Unix/X11 中，一个会话可能包括运行在不同计算机和多个显示器上的应用程序。

2.16.1 会话管理

会话管理包括如下内容。

（1）关闭一个会话

一个会话可以被会话管理器关闭，通常在用户 logout 时关闭。一个系统可以在紧急情况下执行自动关闭，例如：在掉电时。正常关机和掉电关机有很大不同，在正常关机时，用户可能想与应用程序

交互，并确定哪些文件应该保存，哪些应该删除；在掉电关机时，没有时间进行交互，甚至用户不在现场。

（2）不同平台上的协议和支持

Mac OS X和 MS-Windows 对应用程序还没有完全的会话管理，如无法恢复以前的会话。它们支持正常的 logout，在得到用户确认后，应用程序有机会取消进程，这是与 QApplication::comm.itData() 方法相对应的功能。X11 自从升级为 X11R6后，支持完整的会话管理。

（3）让会话管理与 Qt一起工作

通过重载 QApplication::comm itData()来使应用程序参加正常的 logout 处理。如果仅应用在 MS-Windows平台上，则只能提供这个重载了，应用程序最好提供一个如图 2-16 所示的程序关闭对话框。

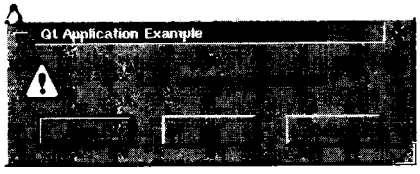


图 2-16　程序关闭对话框

对于完全的会话管理（目前仅 X11R6），程序员还应该关心应用程序的状态存储和下一个会话生命周期恢复状态。

2.16.2　测试和调试会话管理

在 Mac OS X 和 Windows 中，会话管理支持由于这些操作系统本身的这种功能的缺乏而受到限制。为了简单地关闭会话并验证应用程序是否如期望的那样执行，最好启动一个其他应用程序。这个应用程序将随后得到关闭消息，从而允许取消关闭。

在 Unix 中可使用一个支持标准的 X11R6 会话管理，或使用 X 联盟提供的会话管理器 xsm。xsm 是标准 X11R6 安装的一部分，它是一个具有图形界面的会话管理器，可用其来管理会话。

下面是使用 xsm 的一个简单的方法。

（1）运 行 X11R6。

（2）在 home 目录下将创建仅包含下面一行的.xsmstartup 文件：

```
xterm
```

这将告诉 xsm 的 default/failsafe（默认/失败安全)会话仅有 xterm；否 则 xsm将尝试触发包括窗口管理器 twm在内的多个客户，而没有什么用途。

（3）从另一个终端窗口启动 xsm，一个会话管理器和 xterm将出现。xterm 有一个其他 shell 没有的属性，即在 xterm 的 shell 中，SESSION_MANAGER 环境变量指向了刚刚启动的会话管理器。

（4）在新的 xterm 窗口中启动应用程序，应用程序将自动连接到会话管理器，可以使用 ClientList 按钮检查连接是否成功。

注意：当启动或关闭会话管理的客户时，不要保护 ClientList 为打开状态；否则 xsm 可能崩溃。

（5）使用会话管理器的 Checkpoint 和 Shutdown 按钮的不同设置检查应用程序的执行行为，本地保存类型表示客户端应该保存其状态，它对应 QApplication::saveState()函数；全局保存类型请求应用程序保存其未存储的变化到永久的全局可访问的存储中，它触发 QApplication::commitData()。

（6）在用户桌面上 xsm 是一个有用的会话管理器，作为一个测试环境，它是稳定而有用的。

2.17 调试技术

Qt 应用程序的调试可以通过 DDD（Data Display Debugger）进行跟踪调试并打印各种调试或警告信息。DDD 是使用 gdb 调试工具的图形工具，它安装在 Linux 操作系统中，使用方法可参考 DDD 的帮助文档。本节说明如何打印各种调试或警告信息。

2.17.1 命令行参数

当运行 Qt 应用程序时，可以指定多个命令行参数来帮助调试，参数说明如下。

（1）-nograb：应用程序不再捕获鼠标或者键盘。当程序在 Linux 中运行 gdb 调试器时，这个选项是默认的。

（2）-dograb：忽略任何隐含或明显的 -nograb，即使 -nograb 出现在命令行的最后，-dograb 也会超过 -nograb 而生效。

（3）-sync：在 X 同步模式下运行应用程序，同步模式强迫 X 服务器立即执行每一个 X 客户端的请求，而不使用缓存优化，它使得程序更加容易测试并且通常会更慢。-sync 模式只对 X11 版本的 Qt 有效。

2.17.2 打印警告和调试消息

Qt 使用 3 个全局函数 qDebug、qWarning 和 qFatal 来打印警告和调试信息到标准错误输出 stderr（在默认情况下为显示屏，也可指定为文件），说明如下。

（1）qDebug() 用来打印调试信息，在调试版本中输出信息；在发布版本中，函数将不起作用。

（2）qWarning()：用来在程序发生错误时打印警告信息。

（3）qFatal()：用来打印致命错误消息并且退出。

这些函数的 Qt 实现在 Unix/X11 中把文本打印到标准错误输出（stderr），在 Windows 中会打印到调试器，可以通过安装一个消息处理器 qInstallMsgHandler()来接收这些函数。

因为 3 个函数的实现类似，因此只分析函数 qDebug。qDebug 函数的参数格式与函数 printf 类似，打印格式化字符串。qDebug 函数（在 src/tools/qglobal.cpp 中）列出如下：

```
static QtMsgHandler handler = 0;        //指向用户定义的打印输出函数的句柄。
static const int QT_BUFFER_LENGTH = 8196;     //内部 buffer 长度

void qDebug( const char *msg, ... )          //msg 格式化的需要打印的字符串
{
   char buf[QT_BUFFER_LENGTH];
   va_list ap;
   va_start( ap, msg );             //使用可变的参数链表
#if defined(QT_VSNPRINTF)
   QT_VSNPRINTF( buf, QT_BUFFER_LENGTH, msg, ap );
#else
   vsprintf( buf, msg, ap );  //将需要打印的信息放入到 buf 中
#endif
   va_end( ap );
   if ( handler ) {//如果用户指定的输出函数存在，使用它来输出信息
     (*handler)( QtDebugMsg, buf );
   } else {
#if defined(Q_CC_MWERKS)
      mac_default_handler(buf); //mac 系统下的默认输出函数
#elif defined(Q_OS_TEMP)
```

```
    QString fstr( buf );
    OutputDebugString( (fstr + "\n").ucs2() );
#else
    fprintf( stderr, "%s\n", buf );        // 输出到stderr
#endif
  }
}
```

在 src/tools/qglobal.h 中定义了 QtMsgHandler 的函数类型，并将函数 qInstallMsgHandler 定义为从动态库中输出函数名，两个定义列出如下：

```
typedef void (*QtMsgHandler)(QtMsgType, const char *);
// Q_EXPORT 表示动态库中输出这个函数名
Q_EXPORT QtMsgHandler qInstallMsgHandler( QtMsgHandler );
```

函数 qInstallMsgHandler 被用户用来定义一个安装处理函数，并返回以前定义的消息处理函数的指针，在一个应用程序中只能定义一个消息处理函数。恢复以前的消息处理函数时，调用 qInstallMsgHandler(0) 函数（在 src/tools/qglobal.cpp 中），列出如下：

```
QtMsgHandler qInstallMsgHandler( QtMsgHandler h )
{
   QtMsgHandler old = handler;
   handler = h;
   return old;
}
```

示例：应用 qInstallMsgHandler

下面的例子说明如何在一个应用程序中安装自己的程序运行信息输出函数，这个例子先定义了信息输出函数 myMessageOutput ，然后在程序的 main 函数中安装信息输出函数。当这个应用函数运行时，就会使用函数 myMessageOutput 输出运行信息。代码如下：

```
#include <qapplication.h>
#include <stdio.h>
#include <stdlib.h>

void myMessageOutput( QtMsgType type, const char *msg )//定义信息输出函数
{
   switch ( type ) {
            case QtDebugMsg: //输出调试信息
               fprintf( stderr, "Debug: %s\n", msg );
               break;
            case QtWarningMsg: //输出警告信息
               fprintf( stderr, "Warning: %s\n", msg );
               break;
            case QtFatalMsg: //输出致命信息
               fprintf( stderr, "Fatal: %s\n", msg );
               abort();                    //中断运行，退出程序
   }
}

int main( int argc, char **argv )
{
   qInstallMsgHandler( myMessageOutput );  //安装信息输出函数
   QApplication a( argc, argv );
   ...
   return a.exec();
}
```

还有另外两个打印对象信息的调试函数，即 QObject::dumpObjectTree()和 QObject::dump

ObjectInfo()。它们只在程序调试版本中输出信息，在发布版本中这两个函数不起作用。函数 QObject::dumpObjectInfo()打印一个对象信号连接等方面的信息，函数 QObject::dumpObjectTree() 打印子对象树。

2.17.3 调试宏

在程序运行中还经常使用宏 Q_ASSERT 和 Q_CHECK_PTR 来输出信息，说明如下。

（1）Q_ASSERT(b) 中的 b 是一个布尔表达式，当 b 是 FALSE 的时候，打印的警告信息类似“ASSERT: ‘b’ in file file.cpp (234)”。

（2）Q_CHECK_PTR(p)中的 p 是一个指针，如果 p 是空的话，打印的警告信息类似“ In file file.cpp, line 234: Out of memory”。

宏 Q_ASSERT（在 src/tools/qglobal.h 中）实质上是调用函数 qFatal 或 qWarning 输出信息，列出如下：

```
#if !defined(Q_ASSERT)
#  if defined(QT_CHECK_STATE)
#    if defined(QT_FATAL_ASSERT)
#      define Q_ASSERT(x)  ((x) ? (void)0 : qFatal("ASSERT: \"%s\" in %s (%d)",#x,__FILE__,__LINE__)) //
打印x，文件名，在程序源代码中的行号
#    else
#      define Q_ASSERT(x)  ((x) ? (void)0 : qWarning("ASSERT: \"%s\" in %s (%d)",#x,__FILE__,__LINE__))
#    endif
#  else
#    define Q_ASSERT(x)
#  endif
#endif
```

宏 Q_CHECK_PTR（在 src/tools/qglobal.h 中）实质上调用函数 qWarning 输出信息，列出如下：

```
#if defined(QT_CHECK_NULL)
#  define Q_CHECK_PTR(p)  (qt_check_pointer((p)==0,__FILE__,__LINE__))
#else
#  define Q_CHECK_PTR(p)
#endif

Q_EXPORT bool qt_check_pointer( bool c, const char *, int );
```

函数 qt_check_pointer （在 src/tools/qglobal.cpp 中）实现信息输出操作，列出如下：

```
bool qt_check_pointer( bool c, const char *n, int l )
{
    if ( c )
      qWarning( "In file %s, line %d: Out of memory", n, l );
    return TRUE;
}
```

示例：运行宏 Q_ASSERT 和 Q_ASSERT

宏 Q_ASSERT 和 Q_ASSERT 常用来检测程序错误，下面的例子使用了这两个宏：

```
char *alloc( int size )
{
    Q_ASSERT( size > 0 ); //如果size > 0表达式不成立，打印警告信息
    char *p = new char[size];
    Q_CHECK_PTR( p ); //如果指针p为空，打印警告信息
    return p;
}
```

Qt 基于不同的调试标记打印不同类型的警告信息， Qt 使用下面的宏定义（在 src/tools/qglobal.h 中）说明了不同的调试标记：

（1）QT_CHECK_STATE：检测一致的/期望的对象状态。
（2）QT_CHECK_RANGE：检测变量范围错误。
（3）QT_CHECK_NULL：检测危险的空指针。
（4）QT_CHECK_MATH：检测危险的数学运算，如被 0 除。
（5）QT_NO_CHECK：关闭所有的 QT_CHECK_...标记。
（6）QT_DEBUG：使调试代码生效。
（7）QT_NO_DEBUG：关闭 QT_DEBUG 标记。

默认情况下，QT_DEBUG 和所有的 QT_CHECK 标记都是打开的。如果要关闭 QT_DEBUG，则定义 QT_NO_DEBUG；如果要关闭 QT_CHECK 标记，则定义 QT_NO_CHECK。

示例：打印不同类型的警告信息

下面的例子根据不同的宏定义打印不同类型的警告信息：

```
void f( char *p, int i )
{
#if defined(QT_CHECK_NULL) //检测危险的空指针
     if ( p == 0 )
        qWarning( "f: Null pointer not allowed" );
#endif

#if defined(QT_CHECK_RANGE) //检测变量范围错误
     if ( i < 0 )
        qWarning( "f: The index cannot be negative" );
#endif
}
```

第 3 章　KDE 窗口系统

KDE 是 UNIX 操作系统中的一个网络透明且易于使用的现代桌面环境，是一个开放的平台，它遵循 GPL License。

K 桌面环境主要提供了 Konqueror 浏览器、KOffice 办公套件、KIO 网络透明、aRts（the Analog Realtime Synthesizer）多媒体体系和桌面环境用户定制，包括使用 X Window 图形界面进行操作所需要的各种应用程序，如窗口管理器（KWin）、网页浏览器、文件管理器、窗口管理器、帮助系统、配置系统和一些实用程序及工具。Konqueror 是 KDE 的网页浏览器、文件管理器及文档阅读器；KIO 网络透明提供了对本地与远程文件的透明访问，而不必关心文件的位置；aRts 多媒体体系是基于 aRts 的多媒体体系，可以播放本地或网络的多媒体音频和视频流。

K 桌面环境严格符合 GNU/Linux 桌面标准，KWin 遵循窗口管理器规范（Window Manager Specification）。Konqueror 遵循桌面登录标准（Desktop Entry Standard），拖放操作遵循 XDrag-and-Drop 拖放（XDND）协议，会话管理遵循 X11R6 会话管理协议（XSMP）。

KDE 开发环境为开发者提供了桌面通信协议（DCOP）、I/O 库（KIO）、组件模型（Kparts）、基于 XML 的 GUI 类及符合标准的 HTML 解释引擎（KHTML），这些技术及相应接口使用方便，简化了开发者的编程工作。

KDE 使用 Qt/X11 多平台 C++工具，其开发工具有 KDevelop 和 Qt Designer。KDevelop 是开发 KDE 和 Qt 的 C++应用程序的 IDE 工具，它包括一个集成的调试器、一个语法高亮的编辑器，使用项目向导处理基本菜单的创建、automake/autoconf 文件，甚至于创建类文档；Qt Designer 是 Qt 的用户界面设计器，它支持版面设计及检查 accelerator 冲突等，用来帮助设计 KDE 应用程序的对话框。

KDE 软件包由各种应用程序及类库组成，代码量很大，KDE 的网站 http://www.kde.org 上提供了详细的类库帮助说明。本章只介绍了 KDE 的核心技术（在 kde-base 软件包中）。

本章介绍了 KDE 的 MIME 类型处理、KDE 服务、KIO 框架、组件框架及国际化等，还分析了 KDE 的启动过程及 kicker 应用程序。

3.1　KDE 3.5 源代码说明

KDE 3.5 源代码在 http://developer.kde.org 网站上可以下载，KDE 3.5 程序包组成说明如表 3-1 所示。

表 3-1　KDE 3.5 程序包组成说明

目录名	内容说明
kdelibs	组成 KDE 的核心库和基本的应用程序，其他的 KDE 模块使用这个模块的 API
kdebase	一些基本的应用程序，包括窗口管理器
kdepim	KDE PIM（Personal Information Management） 资源、数据处理，以及在 Kontact suite 中的所有应用程序
koffice	KOffice 应用程序
kdegames	KDE 游戏

续表

目录名	内容说明
kdeedu	KDE 教育软件
kdenetwork	KDE 风格应用程序
kdeutils	KDE 实用工具应用程序
kdemultimedia	KDE 多媒体应用程序
extragear-libs	Extragear 的库
extragear-graphics	Extragear 绘图相关的应用程序
extragear-network	Extragear 网络应用程序
playground-base	Playground 的基本应用程序
playground-graphics	Playground 绘图相关的应用程序
playground-sysadmin	Playground 系统管理相关的应用程序

在 kde-base 程序包中的应用程序功能说明如表 3-2 所示。

表 3-2 kde-base 程序包中的应用程序功能说明

文件夹	应用程序功能说明
kdesktop	处理桌面图标，弹出菜单和屏保系统的程序
kaddressbook	一个独立的地址簿
kappfinder	把非 KDE 程序添加到 K-menu（K 菜单）的工具，kappfinder 将非 KDE 应用程序使用的.desktop 文件安装到 KDE 的 applnk 目录中
kcontrol	KDE 控制中心，用来设置 KDE
kdeprint	管理打印工作
kdm	KDE 显示管理器（xdm 的一个替代品），管理登录界面
Kfind	find 工具的图形化
khelpcenter	kde 帮助工具
khotkeys	处理 KDE 的快捷键
kicker	KDE 面板
klipper	一个剪切板查看工具
kmenuedit	整理和添加 K-menu 的工具
konqueror	文件管理器和网页浏览器
konsole	高度可定制的 X 终端模拟器
kscreensaver	设置屏保的程序
ktip	一个提供 KDE 信息和小技巧的向导
kate	KDE 桌面的一个快速文本编辑器（texteditor），它基于 Emacs 样式文档接口
kcheckpass	KDE 的授权检查程序
kcminit	为控制模块运行启动初始化
kdcop	DCOP 的浏览器/执行器，用于查找系统中 DCOP 客户端及其接口和接口的参数
kdebugdialog	在—fullmode 中，可选择将调试输出转储到一个文件
kdepasswd	Unix 的 passwd 命令的 KDE 前端
kdesu	运行 Unix 的 su 命令的 KDE 图形化前端，它允许以其他用户和密码运行应用程序
kdialog	允许从 shell 脚本显示对话框
knetattach	网络向导

续表

文件夹	应用程序功能说明
kpager	桌面回顾，一种浏览桌面功能的说明程序
KPersonalizer	桌面外观的个性化设置应用程序
ksmserver	KDE 的新会话管理服务器
ksplashml	用来显示 KDE 飞溅的屏幕
kstart	使用特殊窗口属性启动应用程序的工具，如图标化、最大化等
ksysguard	使用 KSysGuard 后台进程监视系统各种要素
ksystraycmd	允许运行任何在系统托盘中的应用程序
kwin	KDE 的窗口管理器
kxkb	KDE 键盘工具

用以下命令安装 kdebase：

```
./configure --prefix=/opt/kde-3.1 --disable-debug \
    --disable-dependency-tracking --disable-lametest --with-shadow &&
make &&
make install
```

3.2 KDE 类库

KDE 类库提供了强大的类支持应用程序编程，这些类按层次分为 kdecore、kdeui 和 khtml 等类库，kdecore 和 kdeui 类库是编写应用程序中常用的两个类库。KDE 在网站 api.kde.org 上为每个类提供了详细的在线说明文档，下面说明每个类库的内容。

（1）kdecore 类库

kdecore 类库提供了与用户界面不相关的 KDE 核心类，所有的 KDE 应用程序使用这个库提供基本的功能，如配置系统、IPC、国际化有本地化支持，以及对文件系统的不依赖于位置的访问等。所有的应用程序都应该连接到 kdecore 库，并使用从 QApplication 派生的类 KApplication，以便使用 KDE 环境。

（2）kdeui 类库

kdeui 类库提供了诸如窗口部件的 KDE 用户界面类，它为 KDE 应用程序提供了标准的用户界面元素。如果你的 KDE 应用程序有 GUI 界面，则需要连接到 libkdeui 库。大多数带有一个或多个顶层窗口部件的应用程序应该使用这个库中的 KMainWindow 类，该类自动提供如会话管理和简单的工具栏/菜单栏/状态栏等特征。并且提供了拼写检查库，KSpell 提供了对国际化的 ISpell 或 Aspell（根据用户的选项）的访问，还有为拼写检查相关的 GUI 界面（如“Add”和“Replace”等）。

可使用 KSpell 自动拼写检查 ASCII 文件、应用在线拼写检查、设定拼写检查格式和进行标识文档（如 HTML 及 TeX 等），这 3 种方法的相关函数分别是 check()、checkWord()和 checkList()。

KspellConfig 保存选项对话框中设置的配置信息，KSpell 通常异步工作。如果不需要异步工作，则应该简单地使用 KSpell::modalCheck()。

（3）khtml 类库

khtml 类库提供了 KDE HTML 组件并及完全特征的 HTML 语法分析器和窗口部件，KDE 应用程序使用它来进行 HTML 浏览，其中包括 Konqueror 浏览器/文件管理器、KMail 和 KDE 帮助系统。这个库还提供了 HTML4/XHTML 库的支持、嵌入 Java 小应用程序、层叠模式的文档页（cascading style sheets （CSS））和 JavaScript。

如果需要添加简单文本浏览的窗口部件到应用程序，则可以使用 kdeui 库中的 KTextBrowser。

（4）kjs 类库

kjs 类库提供了 Javascript （支持 aka、ECMAScript 和 Jscript）.

（5）kio 类库

kio 类库提供了对网络文件的低层次访问类，以及设备的访问类，如 KDirWatcher 类监视目录的变化。

（6）kparts 类 库

kparts 类库支持可重用、嵌入且可扩展的应用程序组件。

（7）dcop 类 库

dcop 类库是 DCOP 通信库。提供了进程间通信的类。

（8）kdefx 类库

kdefx 类库提供了位图和图像操作的类。

（9）kmdi 类 库

kmdi 类库提供了多文档接口相关的类。

（10）libkmid 类库

libkmid 类库提供了与 Midi 操作相关的类。

（11）kdeprint 类库

kdeprint 类库提供了高层次打印机控制功能类。

（12）interfaces 类库

interfaces 类库提供了接口类定义通用组件的接口，以便新的应用能被访问。

（13）kabc 类库

kabc 类库提供了访问 KDE 地址簿的类。

（14）kresources 类库

kresources 类库提供了 KDE 资源系统管理及操作的类。

（15）kutils 类库

kutils 类库提供了高层次的实用工具相关的类，如搜索/替换功能的支持。

（16）arts 类库

arts 类库提供了声音播放等方面的类。

（17）knewstuff 类库

knewstuff 类库提供了与应用程序数据上载和下载的相关类。

（18）dnssd 类库

dnssd 类库提供了网络服务出版及服务查询方面的类。

（19）kunittest 类库

kunittest 类库提供了 KDE 中的单元测试框架方面的类。

3.3 系统资源访问

3.3.1 标准资源目录

KDE 为不同的资源类型提供了不同的目录， KDE 提供的资源类型说明如表 3-3 所示。

表 3-3 KDE 提供的资源类型说明

资源类型	说　明
apps	应用程序菜单（用.desktop 文件）
cgi	从 khelpcenter 运行的 CGI
config	配置文件
data	应用程序存储数据处
exe	KDE 使用的私有安装的可执行应用程序
html	HTML 文档
icon	出现在窗口管理器或面板上的应用程序图标
lib	动态库，将成为被 dlopen 打开的模
locale	KLocale 翻译文件
mime	mime 类型
sound	应用程序声音
toolbar	工具栏图片
wallpaper	墙纸

所有这些资源存在于源代码中，在源代码编译时被 configure 配置脚本拷贝到指定的安装目录下。

KDE 提供了类 KStandardDirs 对标准资源类型目录进行管理，应用程序不需要知道资源在硬盘上的位置，而只需通过类 KStandardDirs 对某种类型资源进行访问，对资源的管理细节由类 KStandardDirs 完成。

3.3.2 KDE 系统配 置缓冲

Sycoca 是 KDE 系统配置缓冲 System Configuration Cache 的缩写，它是 kregistry 的继任者。作为一个轻量级数据库，对静态系统配置信息快速查找和最小内存使用做了优化。

这个数据库将提供多个客户端的仅读访问，数据库被 kbuildsyscoca 可读的配置文件创建或更新。kded监视配置文件，并且在任何配置文件的更改被检测到时，调用 kbuildsycoca 更新数据库。

更新数据库意味着数据库整个被一个更新的版本替代。在查找静态系统信息时，通过 Sycoca，而不需要分析多个文件。应用程序从不直接使用 Sycoca，而是通过 KDE 提供了类来访问。通过使用表 3-4 所示的类获得系统信息。

表 3-4 应用程序中用来访问 Sycoca 库的类

类　名	功能说明
KImageIOFormat	提供 KDE 支持的图像文件格式信息
KProtocolInfo	提供了 KDE 支持的协议信息，这些协议被使用在 URL 中，并将映射到一个协议支持的 IO slave
KServiceGroup	提供了服务的链表，如 K 菜单中应用程序的层次、控制中心设置的层次或屏幕保护的链表（控制中心的屏幕保护设置）
KService	提供了一个服务的信息，一个服务总被一个.desktop 文件描述
KServiceType	描述了能被一个服务提供的服务类型，如它的子类 KMimeType 表示打开一特定格式文件的能力。如一个 HTML 文件的 MIME 类型是 text/html，一个或多个服务能打开这个文件（如 konqueror、netscape 和 kwrite）

3.4 图形支持

Qt 的低层次图像模型基于 X11 提供的能力和 Qt 引入的其他窗口系统，它还可通过位图和文本的

变形得到扩展。Qt 基本的 2D 绘制图形类是 QPainter，它能在 QPaintDevice 上绘制。QPainter 提供了基本的图形绘制函数，这些函数还因参数不同而提供不同的重载版本。有 3 种可能的绘制设备使用，一是 QPainterDevice，它代表屏幕上的一个窗口部件；二是 QPrinter，它代表一台打印机，并可产生 Postscript 输出；三是 QPicture，它记录绘制命令，并保存在硬盘中，以后再回放这些命令。绘制命令的可能存储格式是 W3C 标准 SVG。

在某些特征的支持下，重复使用用于显示一个窗口部件的着色代码来打印是可能的。绘制一个窗口部件在窗口 部件类的 paintEvent() 方法中进行，paintEvent() 函数类似如下代码：

```
void FooWidget::paintEvent()
{
   QPainter p(this);
   // 建立 painter
   // 使用 painter
}
```

当在打印机上绘制时，必须使用 QPrinter::newPage()在一页上完成绘制，它与窗口绘制毫无关系。

3D 图形的标准是 OpenGL，其本身仅通过 GL 上下文与帧缓冲的特定区域上的着色打交道，与环境工具没有任何交互。Qt 提供了窗口部件 QGLWidget，它用一个相关的 GL 上下文封装了一个窗口。基本上，通过从它派生类并重用它的一些方法来使用 QGLWidget。

代替重用 paintEvent() 和使用 QPainter 绘制一个窗口部件的内容，应重载 paintGL() 并使用 GL 命令着色画面。在 paintGL()被调用之前，QLWidget 将产生当前使用的 GL 上下文，并且它在 paintGL()被调用之后刷新画面。

在第 1 次 resizeGL()或 paintGL()被调用之前，虚拟方法 initializeGL()被调用，这可被用来控制对象的显示列表，并做任何初始化。

代替重用 resizeEvent()，可重载 resizeGL()，这可被用来设置合适的 viewport。

当画面的状态有变化时，代替调用 update()，应该调用 updateGL()触发重画事件。

整体来说，QGLWidget行为像其他的窗口部件，如可以像通常一样处理鼠标事件，并重定义窗口部件大小。

OpenGL 是一个 3D 图形的低层接口，Qcanvas 为编程者提供了对象及其属性的一个高层次接口。同样，也有高层次的 3D 图形接口，比较出名的有 SGI 开发的 Open Inventor. Originally，还有开放源代码的绑定在 Qt 上的 SoQt 工具。

3.5 用户界面

3.5.1 Action 模式

创建用户界面通常意味着调用各种菜单、工具和状态栏对象的多个 insertItem() 和 insertButton() 方法。有时，个条目被显示在多个控件 上，尽管图标不同，但其动作（如：激活）代表的操作是一样的。如标准编辑 UI 条目“ cut”、“copy ”和 “paste”，它们显示在工具栏、“ 编译 ”菜单和弹出菜单中。

同一操作与多个 UI 容器对象相关联，KDE Action 类解决这一问题。开发者分配一个 KAction 对象，并将对象的 activation 信号与 KAction对象连接，具体使用方法请参看 KDE 的在线文档 KAction 类说明。

3.5.2 用 XML 定义菜单和工具栏

由于在没有改变源代码的情况下定制菜单等控件是困难的，所以为了解决这个问题，可以使用称

为“ XMLGUI ”的一套类。基本原理是将控件（如菜单栏和工具栏）的外观与动作分开，控件外观用 XML 编写；动作使用 C++编程。不用修改任何源代码，控件可简单地通过调整 XML 文件来改变。XMLGUI 对于模块化程序来说是很重要的，出现在菜单栏中的条目可能来自许多不同的插件或控件。

KDE 顶层窗口的类 KMainWindow 从 KXMLGUIClient 类继承，因此它支持 XMLGUI ，在类 KMainWindow 中创建的所有动作必须以 KXMLGUIClient 的 actionCollection() 作为父亲。一个对 createGUI()的调用将构建定义在应用程序 XML文件（以 rc 结尾）的整个菜单和工具栏中。

3.5.3 在线帮助

KDE提供了多种帮助方式，分别说明如下。

（1）Tooltips（工具提示）

Tooltips 是当用户移动鼠标到提供 Tooltips 的组件上停留一段时间后出现的弹出式帮助信息，当鼠标移走时信息消失。Tooltips 对工具栏等特别有用，也足够用来说明按钮的用途。

（2）What's this?

“What's this?”帮助是用一个窗口部件或菜单项提供帮助说明。在对话框中，它可通过两种方式被激活，即通过按 Shift+F1 键或在标题栏中点击问题标识。然后鼠标变成问题标识形状的箭头，当用户点击窗口部件中的一个元素时，“What's this?”帮助带有菜单项对话框被打开，并且相关联的帮助条目被打开。

（3）手册（Handbook）

每个应用程序有一本手册，该手册一般通过激活帮助菜单后在 khelpcenter 中被查看。KDE帮助手册是使用 DocBook标识语言编写的。

3.5.4 复杂窗口部件

KDE 提供了各种窗口部件类，以及这些窗口部件类的详细说明，这里不再说明。 KDE 还提供了复杂的窗口部件，如 KHTML和文件对话框等。下面说明几个常用的复杂窗口部件。

（1）KHTML

KHTML 是一个 XML/HTML4 相适应的 HTML 库，它支持 DOM、Java、JavaScript 和 CSS（Cascading Style Sheet 层叠样式表）。在程序中使用 KHTML很容易，通过 KHTMWidget类的成员函数，可以完成 KHTML 的许多功能，如可以通过 KHTMLWidget::document() 方法访问被显示的文档。KHTML 提供了 DOM（文档对象模型 Document Object Model）在第 1 级和第 2 级别的完整应用，DOM 是使用内部的类持有文档数据的应用。

（2）文件对话框库

KDE 文件对话框被应用在 kfile 库中，这个库还包括其他的窗口部件。与文件对话框相关的类实际上是一个文件对话框构造函数集，如它不仅有传统上用来为打开或存储获得文件名的 KFileDialog 类，还提供了分开使用的各种单独的窗口部件。

（3）显示大量数据的类

使用 QListView, QListBox 和 QIconView 类可以显示大量的数据。

3.6 MIME 类型处理

MIME 类型被用来描述文件或数据块的内容类型，最初它们是为了用 E-mail 发送图像和声音文件而引入的。MIME 的含义是多用途 Internet 网邮件扩展，即 “Multipurpose Internet Mail Extensions ”的

缩写。后来这个系统也被网页浏览器用来决定如何打开网页服务器送给用户的数据，例如一个 HTML 页有一个 MIME类型“text/html”，即“应用程序/后缀”。在 KDE 中，MIME 类型被用到各种地方，下面列出 4个应用例子。

（1）在 Konqueror 的图标视图（icon view）中，文件用图标表示，每种 MIME 类型用一种图标显示。

（2）在 Konqueror 中点击一个文件图标或文件名时，显示视图中的文件或与打开这个文件类型相关的应用程序。

（3）当从一个应用程序拖放一些数据到另一个应用程序（或在同一个应用程序内）时，放下的目标可能仅选择接收某一类型的数据，并且将根据不同的文本数据处理图像数据。

（4）剪贴板有一个 MIME 类型，通常 X 程序仅处理位图或文本，但在 Qt 中没有数据类型限制。

MIME 类型建立的过程是首先必须建立从文件名到 MIME 类型的映射，在文件不可用的情况下，文件内容也可映射到 MIME 类型。然后必须映射 MIME 类型到查看或编辑某种类型文件的应用程序或库，或者为其创建一个图标。

MIME 类型的使用方法是一个文档或代表文档的图标与一个 MIME 类型对应，一个 MIME 类型与一个或多个应用程序对应，这样一个文档可以找到一个或多个应用程序打开它。当有多个应用程序可以打开这个文档时，使用优先级最高的应用程序。 MIME 类型的使用方法如图 3-1 所示。

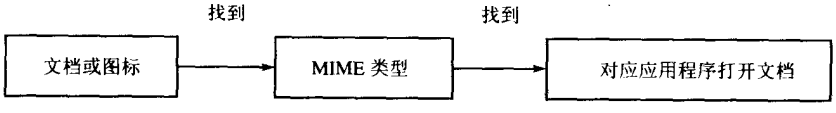


图 3-1 MIME类型的使用方法

3.6.1 一个应用 MIME 类型的例子

每个 MIME 类型用一个.desktop 文件描述，说明 MIME 类型、图标及打开的应用程序、 Comment 等。下面以一个例子说明 MIME类型对应的.desktop 文件的定义、安装及程序获得 .desktop信息方法。

示例：MIME 类型定义及应用

为一个新的 foobar 程序定义一个类型“application/vnd.kde.foo”，如果所用的 MIME 类型没有注册，则还应向 IANA 组织注册。

必须写一个文件 vnd.kde.foo.desktop 描述 MIME 类型相关信息，并把它安装在${KDEDIR}/share/mimelnk/application 中。为此，必须在 Makefile.am 中添加相关安装语句才能在编译安装时将.desktop 文件拷贝到安装目录下。Makefile.am 中添加的语句列出如下：

```
mimedir = $(kde_mimedir)/application
mime_DATA = vnd.kde.foo.desktop
EXTRA_DIST = $(mime_DATA)
```

文件 foo.desktop 的内容如下：

```
[Desktop Entry]
Type=MimeType
MimeType=application/vnd.kde.foo  #定义 MIME 类型
Icon=fooicon    #代表这个 MIME 类型的图标
Patterns=*.foo;  #MIME 类型的模式，表示以.foo 结尾的文件是这种 MIME 类型
DefaultApp=foobar    #默认的打开应用程序
Comment=Foo Data File
Comment[de]=Foo Datei
```

在 KDE 程序库中，文件 foo.desktop 中的内容被映射到一个 KMimeType类实例。这样，用户程序只需要访问 KMimeType 类实例即可知道.desktop 文件的内容，而不必关心.desktop 文件的存储位置。可以如下使用实例打印文件 foo.desktop 信息：

```
KMimeType::Ptr type = KMimeType::mimeType("application/vnd.kde.foo");
kdDebug() << "Type:    " << type->name() < endl;  // kdDebug()表示输出调试信息到终端
kdDebug() << "Icon:    " << type->icon() < endl;
kdDebug() << "Comment: " << type->comment() < endl;
QStringList patterns = type->patterns();
QStringList::ConstIterator it;
for (it = patterns.begin(); it != patterns.end(); ++it)
kdDebug() << "Pattern: " << (*it) << endl;
```

如果需要从文件的内容，而不是文件名找出 MIME 类型，这是更可靠，但也较慢的方法。因为它需要读取部分文件内容，这是通过类 KMimeMagic 类来实现的。方法如下：

```
KMimeMagicResult *result = KMimeMagic::self()->findFileType("/home/bernd/foobar.jpg");
if (!result || !result->isValid())
   kdDebug() << "Could not find out type" << endl;
else
   kdDebug() << "Type: " << result->mimeType() << endl;
```

当然，KMimeMagic 仅能从本地文件的内容中决定文件类型。对于远程文件，可使用如下方法得到文件类型：

```
KURL url("http://developer.kde.org/favicon.ico");
QString type = KIO::NetAccess::mimetype(url);
if (type == KMimeType::defaultMimeType())
   kdDebug() << "Could not find out type" << endl;
else
   kdDebug() << "Type: " << type << endl;
```

上面这个例子开始了一个 KIO 工作下载文件的一部分，并检查文件类型，这个函数可能较慢并且阻塞程序运行。正常情况下，如果 KMimeType::findByURL() 已返回“application/octet-stream”，则才可以使用它来得到 MIME 类型。

另一方面，如果不想阻塞应用程序，则也能开始 KIO 工作并且连接其一些信号，方法如下：

```
void FooClass::findType()
{
     KURL url("http://developer.kde.org/favicon.ico");
     KIO::MimetypeJob *job = KIO::mimetype(url);
     connect( job, SIGNAL(result(KIO::Job*)),
               this, SLOT(mimeResult(KIO::Job*)) );
}

void FooClass::mimeResult(KIO::Job *job)
{
     if (job->error())
          job->showErrorDialog();
     else
          kdDebug() << "MIME type: " << ((KIO::MimetypeJob *)job)->mimetype() << endl;
}
```

3.6.2 映射 MIME 类型到一个应用程序或服务

当安装一个应用程序时，同时安装一个含有这个应用程序 MIME 类型信息的.desktop 文件。类似地，诸如 KParts 一样的组件通过其服务描述文件.desktop 使组件信息可用。通常，有多个程序和组件

能处理个给定的 MIME 类型。可从 KServiceTypeProfile 类获得一个服务提供链表，方法如下：

```
KService::OfferList offers = KServiceTypeProfile::offers("text/html", "Application");
Kservice::OfferList::ConstIterator it;
for (it = offers.begin(); it != offers.end(); ++it) {
  Kservice::Ptr service = (*it);
kdDebug() << "Name: " << service->name() << endl;
}
```

这个函数返回服务提供的链表，一个 KserviceOffer 包括一个 KService::Ptr 和一个优先级数，KServiceTypeProfile::offers()返回的链表按用户的优先数排序。用户通过在 Konqueror 的 HTML 文件的上下文菜单调用‘ keditfiletype text/html ”或选择“Edit File Type” 可以改变这个顺序。

在大多数情况下，为 MIME 类型和服务类型提供所有服务的链表并不重要，可以使用下面的函数得到最高优先级的服务：

```
KService::Ptr offer = KServiceTypeProfile::preferredService("text/html", "Application");
if (offer)
   kdDebug() << "Name: " << service->name() << endl;
else
   kdDebug() << "No appropriate service found" << endl;
```

3.7 KDE 服务

服务是 KDE 模块化结构的核心概念，它可能是共享库形式的插件，或通过 DCOP 控制的程序。通过声明某一服务类型，一个服务使用某一 API 或特征。

这种将服务与应用程序分开的好处是明显的，一个使用一个服务类型的应用程序不必知道它需要使用的具体插件或 API，而仅需要使用与这种服务类型相关的插件或 API。这样，提供服务类型的插件或 API 能够在不影响应用程序的情况下被改变。还有，用户还能为某一特征（即服务类型）配置服务。

为了说明服务类型的概念，下面列出了一些使用服务类型的例子。

（1）用在 Konqueror 中的 HTML 着色引擎是一个应用了 KParts/ReadOnlyPart 和 Browser/Vie 服务类型的嵌入式组件。

（2）在 KDevelop HEAD 中，大多数功能被打包在带有 KDevelop/Part 服务类型的插件中。在启动时，所有与这个类型相关的所有服务被装载，这样可弹性地扩展 Kdevelop IDE。

应用程序可通过两种方式使用服务，一是应用程序通过插件实现功能，方法是通过服务类型找到对应提供服务的共享库插件，然后通过类工厂查找到插件的具体功能函数；二是应用程序通过 DCOP 启动另一个应用程序，如在桌面菜单上点击应用程序条目来启动应用程序，方法是通过 DCOP 以应用程序名启动应用程序。应用程序使用服务的两种方式如图 3-2 所示。

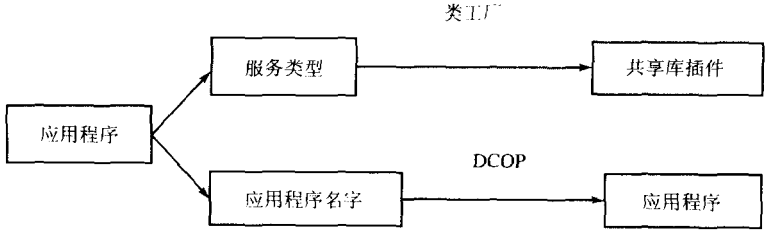


图 3-2 应用程序使用服务的两种方式

3.7.1 定义服务类型

增加新的服务类型是通过安装服务类型的描述到$KDEDIR/share/servicetypes 目录来实现的。在automake框架中，与服务相关的 Makefile.am 中必须添加类似下面的语句安装服务描述文件：

```
kde_servicetypesdir_DATA = kdeveloppart.desktop    #服务类型相关的描述文件
EXTRA_DIST = $(kde_servicetypesdir_DATA)
```

一个 KDevelop组件（part）的 kdeveloppart.desktop 文件内容类似于下面的描述：

```
[Desktop Entry]
Type=ServiceType  #类型为服务类型
X-KDE-ServiceType=KDevelop/Part  #服务类型的定义
Name=KDevelop Part  #服务组件的名字

[PropertyDef::X-KDevelop-Scope]  #属性名
Type=QString  #属性的数据类型

[PropertyDef::X-KDevelop-ProgrammingLanguages]
Type=QStringList

[PropertyDef::X-KDevelop-Args]
Type=QString
```

上面这个例子说明了如何声明一个带有属性的服务，每个属性在文件中用一个组进行描述。组格式是[PropertyDef:: 属性名]. Type 条目声明属性数据类型，属性的所有数据类型在 QVariant 类中定义。

3.7.2 定义共享库服务

服务定义存在$KDEDIR/share/services 目录中，与服务相关的 Makefile.am 中必须添加类似下面的语句安装服务描述文件：

```
kde_servicesdir_DATA = kdevdoxygen.desktop
EXTRA_DIST = $(kde_servicesdir_DATA)
```

服务描述文件 kdevdoxygen.desktop 定义 KDevDoxygen 插件及其服务类型 KDevelop/Part，列出如下：

```
[Desktop Entry]
Type=Service      #表示是服务
Comment=Doxygen  #描述
Name=KDevDoxygen  #服务插件的名字
ServiceTypes=KDevelop/Part    #定义服务类型
X-KDE-Library=libkdevdoxygen  #插件对应的共享库
X-KDevelop-ProgrammingLanguages=C,C++,Java  #编程语言
X-KDevelop-Scope=Project
```

在库中必须包含下面的函数：

```
extern "C" {
   void *init_libkdevdoxygen()
   {
      return new DoxygenFactory;
   }
};
```

工厂类 DoxygenFactory 的类型依赖于服务使用的特定服务类型，在我们的 KDevelop 插件样例中，

类工厂必须是 KDevFactory（从 KLibFactory 继承）。更常用的是 KParts::Factory，它支持产生 KParts::ReadOnlyPart 对象。

为了在一个应用程序中使用一个共享库服务，必须获得代表这个服务的 KService 对象。如果已建立了 KService 对象，则可简单地装载这个库并且得到其工厂对象的指针，例如：

```
KService *service = ...
QString libName = QFile::encodeName(service->library());
KLibFactory *factory = KLibLoader::self()->factory(libName);
if (!factory) {
      QString name = service->name();
      QString errorMessage = KLibLoader::self()->lastErrorMessage();
      KMessageBox::error(0, i18n("There was an error loading service %1.\n"
                                 "The diagnostics from libtool is:\n%2")
                        .arg(name).arg(errorMessage);
}
```

从这时起，更多地处理依赖于服务类型。对于普通插件，使用 KLibFactory::create() 方法创建对象；对于 KParts，必须将工厂指针指向更特殊的 KParts::Factory，并使用它的 create()方法：

```
if (factory->inherits("KParts::Factory")) {
     KParts::Factory *partFactory = static_cast<KParts::Factory*>(factory);
     QObject *obj = partFactory->createPart(parentWidget, widgetName,
                                            parent, name, "KParts::ReadOnlyPart");
   ...
} else {
    cout << "Service does not implement the right factory" << endl;
}
```

3.7.3 定义 DCOP 服务

一个 DCOP 服务通常作为一个程序应用，这个程序在需要的时候被启动，然后它进入事件循环并侦听 DCOP 的连接。程序可能是一个交互的，但也能完成运行或其生命周期的一部分作为不需要用户注意的后台进程（daemon）。例如，一个 kio_uiserver 的后台进程使用了用户交互，如 KIO 库的进度条对话框。这样文件的下载进度能显示在一个进度条对话框中，即使这些下载是从不同的应用程序初始化的。

一个 DCOP 服务的定义不同于共享库服务，它不是指定一个库，而是一个可执行的程序。 DCOP 服务也不指定一个 ServiceType 行，因它通常通过其名字被启动。作为附加属性，它包括下面两项。

（1）X-DCOP-ServiceType

X-DCOP-ServiceType 指定服务开始的方式，这个值仅说明这个服务的启动不能超过一次。这意味着，如果你尝试开始这个服务（如通过 KApplication::startServiceByName()） KDE 查找它是否已用 DCOP 注册，并且是否使用正在运行的服务。如果未注册， KDE 将启动它，并等待直到被注册。这样，可立即发送 DCOP 调用这个服务。在这种情况下，服务应该被作为一个 KUniqueApplication 使用。

X-DCOP-ServiceType 的值为 Multi 时，说明服务的多个实例能够共同存在，这样每个实例尝试开始将创建另一个进程的服务。作为值的一个最后可能性，可使用 None。在这种情况下，服务的启动将不等待直到它被用 DCOP 注册。

（2）X-KDE-StartupNotify

X-KDE-StartupNotify 正常下应被设置 false；否则当程序被启动时，任务栏将显示一个启动警告或依赖于用户的设置，光标将被改变。

下面是 kio_uiserver 的定义：

```
[Desktop Entry]
```

```
Type=Service
Name=kio_uiserver
Exec=kio_uiserver    #应用程序
X-DCOP-ServiceType=Unique
X-KDE-StartupNotify=false
```

下面说明如何使用 DCOP 服务

在 KApplication 中有多种方法可启动一个 DCOP 服务，其中之一列出如下：

```
DCOPClient *client = kapp->dcopClient();
client->attach();
if (!client->isApplicationRegistered("kio_uiserver")) {
     QString error;
   if (KApplication::startServiceByName("kio_uiserver", QStringList(), &error))
        cout << "Starting kioserver failed with message " << error << endl;
}
...
QByteArray data, replyData;
QCString replyType;
QDataStream arg(data, IO_WriteOnly);
arg << true;
if (!client->call("kio_uiserver", "UIServer", "setListMode(bool)",
                          data, replyType, replyData))
    cout << "Call to kio_uiserver failed" << endl;
...
```

3.8 KIO 框架

在网络时代，桌面应用程序访问 Internet 网资源是基本的功能，它应能从网络服务器下载文件，写文件到一个 FTP 服务器或读取网络服务器中的邮件。通常，这种无论文件的位置在哪里都能访问的能力被称为“网络透明”。

旧的 NFS 文件系统是一个在 POSIX API 级别上应用网络透明的尝试，网络透明访问中异步操作是重要的。例如，当你等待网页浏览器下载一页时，用户界面不能被阻塞。还有，网页着色不能到网页完成可用时才开始，而是应该在数据进来时有规则地更新。

在 KDE 库中，网络透明性是通过 KIO API 来实现的。这个框架的核心概念是 IO 工作，一个工作可能是拷贝或删除文件或类似的事情。一旦一个工作开始，则运行在后台并不阻塞应用程序。来自工作的任何通信返回到应用程序，如分发数据或进度条信息。

文件定位由 URL 来描述，KDE 使用相同语法的 URL 扩展来访问资源。URL 不仅扩展了地址的范围到本地文件系统，还能访问压缩的归档文件，如可以浏览 tar 的归档文件。在 HTTP 服务器上，一个 tar 归档文件 article.tgz 中的文件 paper.tex 用 URL 描述如下：

```
http://www-com.physik.hu-berlin.de/~bernd/article.tgz#tar:/paper.tex
```

KIO API 库提供了大量的类，实现了所需要的几乎所有文件管理函数。实际上， KDE 文件管理器和 Konqueror 都使用这些函数提供网络文件管理。从 KDE 应用程序中使用这个库的最简单的方法是通过 KIO::NetAccess 类（较容易的同步访问方式）和 KIO::Job 类（较复杂的异步工作方式），这个库还应用了系统配置缓冲 KSycoca（System Configuration Cache）。

所有的文件操作都通过 kioslave 在后台异步完成，后台操作是通过启动 kioslave 执行某一任务而获得的。kioslave 作为一个单独进程启动，并通过 UNIX domain socket 通信。这样，不需要多线程，并且不稳定的 kioslave 不能崩溃使用其应用程序。 kioslave 支持 smb、system、tftp、ftp 及 file 等协议。通过输入统一的 URL 后，系统会选择不同协议的 kioslave。例如，在 URL 中输入的地址为 file:/ home/

test，系统会选用支持 file 协议的 kioslave 处理，从而显示了这个目录下的文件。

3.8.1 同步使用 KIO

在大多数情况下，通过调用在 KIO 名字空间的函数创建工作。这些函数以一个或两个 URL 作为参数，还加上其他需要的参数。当工作完成时，它发出信号 result(KIO::Job*)。在这个信号发出后，工作删除其自身。个典型的例子如下：

```
void FooClass::makeDirectory()
{
   SimpleJob *job = KIO::mkdir(KURL("file:/home/bernd/kiodir"));
   connect( job, SIGNAL(result(KIO::Job*)),
          this, SLOT(mkdirResult(KIO::Job*)) );
}

void FooClass::mkdirResult(KIO::Job *job)
{
   if (job->error())
      job->showErrorDialog();
   else
      cout << "mkdir went fine" << endl;
}
```

依赖于工作的类型，还可以连接其他信号。 KIO 提供了文件系统中文件和目录的各种操作函数，如 mkdir、rmdir、chmod、rename、symlink 及 file_copy 等。

3.8.2 同步使用 KIO

通常，KIO 的异步 API过于复杂，因此应用全速的异步不是首选。例如，在一个一次仅处理一个文档文件的应用程序中，当程序在下载一个文件时很少要执行其他操作。在这种情况下，可使用一套静态函数中的一个简单 API 来完成。

NetAccess 提供了网络透明、容易且同步的 KIO 文件操作接口，它允许执行简单的文件操作（如 load、save、copy 和 delete 等）而不需要直接使用 KIO::Job。KIO::Job 是异步的，意味着开发者必须用槽函数连接它的信号，KIO::NetAccess 提供了同步的下载、上载和临时文件创建与删除等操作。这个函数出现是阻塞的，但在这个操作被处理时， Qt 事件循环继续运行，这意味着 GUI 不会停止不动。这个类不被用做一个类实例，而是使用静态函数。

例如：拷贝一个文件可使用如下方法：

```
KURL source, target;
source = ...;
target = ...
KIO::NetAccess::copy(source, target);
```

在拷贝进程已完成后这个函数将返回，这个方法提供了一个进度条对话框，确保应用程序处理 repaint 事件。

3.8.3 MetaData

MetaData 类是 key/value 字符串的简单映射，这个类声明如下：

```
class KIO_EXPORT MetaData : public QMap<QString, QString>
```

对 IO 工作的接口很抽象，未考虑任何协议特定的 IO slave 与应用程序之间的信息交换。有时，这种情况是不合适的。例如，可能将某一参数给 HTTP slave 控制缓存的行为或发送一堆带有请求的 cookie。在这种情况下，meta 数据的概念必须被引入。当工作被创建后，可以通过为其增加 meta 数据来配置工作每个 meta 数据条目由 key/value 对组成，例如为了阻止 HTTP slave 从它的缓存中装载一个网页，可以使用如下代码：

```
void FooClass::reloadPage()
{
    KURL url("http://www.kdevelop.org/index.html");
    KIO::TransferJob *job = KIO::get(url, true, false);
    job->addMetaData("cache", "reload");
    ...
}
```

这个技术还被用到其他方面，例如对于从 slave 到应用程序的通信，Job::queryMetaData() 方法请求由 slave 分发的某个 key 对应的 value 下面是一个例子，当网页发生变化时查询网页修改的日期：

```
void FooClass::printModifiedDate()
{
    KURL url("http://developer.kde.org/documentation/kde2arch/index.html");
    KIO::TransferJob *job = KIO::get(url, true, false);
    connect( job, SIGNAL(result(KIO::Job*)),
             this, SLOT(transferResult(KIO::Job*)) );
}

void FooClass::transferResult(KIO::Job *job)
{
    QString mimetype;
    if (job->error())
        job->showErrorDialog();
    else {
        KIO::TransferJob *transferJob = (KIO::TransferJob*) job;
        QString modified = transferJob->queryMetaData("modified");  //查询修改的日期
        cout << "Last modified: " << modified << endl;
}
```

3.8.4 调度

使用 KIO API 通常不必处理启动 IO slave 及与其通信的细节，正常使用的方法是启动一个工作，并处理工作发出的信号。

当创建一个工作时，它被放在一个队列中。当应用程序回到事件循环时，KIO 为在队列中的工作分配 slave 进程。对于第 1 个工作开始来说，还需为合适的协议启动一个 IO slave。然而在这个工作(如从 http 服务器上下载）已完成后，这个工作不是立即被杀死，而是被放进一个空闲 slave 的池中并在某时间的无活动期（当前是 3 分钟）后才被杀死。如果一个同一主机、同一协议的新请求到达，这个 slave 被再次使用，这样节省了创建新工作及建立连接的时间。

（1）Direct

这是默认方式，当创建一个工作时，KIO::Scheduler 将被通知并找到一个正存在的空闲 slave 或者创建一个新的 slave。这种直接指定 slave 给工作的方式称为“direct”。它并不总是最合适的方案，因为它可能需要很多的内存，并且对客户端和服务器都产生高的负载。例如：

```
TransferJob *job = KIO::get(KURL("http://www.kde.org"));
```

（2）调度

在创建很多工作时，可能不想为每个工作创建一个 slave。而调度一个工作时，则创建最大的数量

将是 slave。当更多的工作到达时，它们将排队。当一个 slave 完成一个工作时，它将从队列中指定一个工作，例如：

```
TransferJob *job = KIO::get(KURL("http://www.kde.org"));
KIO::Scheduler::scheduleJob(job);
```

（3）面向连接

对于某些操作来说，多个工作使用同一个连接是重要的。在所有工作使用同一 slave 时，必须使用同个连接。例如，对于 IMAP slave，它对启动同一服务器的多个进程没有差别，同一时间仅一个 IMAP连接应被强制进行。这种情况下，应用程序必须明确地与一个 slave 打交道。它必须为某一连接分配一个 slave，并且指定所有的应该通过同一连接到同一 slave 的工作，可通过使用 KIO::Scheduler 获得。

可请求调度器用面向连接的操作打开一个 slave，然后用这个调度器指定工作给这个 slave。工作将被排队，slave 将依次处理这些工作，例如：

```
Slave *slave = KIO::Scheduler::getConnectedSlave(
        KURL("pop3://bastian:password@mail.kde.org"));
TransferJob *job1 = KIO::get(
        KURL("pop3://bastian:password@mail.kde.org/msg1"));
KIO::Scheduler::assignJobToSlave(slave, job1);
TransferJob *job2 = KIO::get(
        KURL("pop3://bastian:password@mail.kde.org/msg2"));
KIO::Scheduler::assignJobToSlave(slave, job2);
TransferJob *job3 = KIO::get(
        KURL("pop3://bastian:password@mail.kde.org/msg3"));
KIO::Scheduler::assignJobToSlave(slave, job3);

// ... 等待工作完成...
KIO::Scheduler::disconnectSlave(slave);
```

注意在连接被破坏时，必须明确地断开 slave 的连接。因此错误处理函数中应包括下面的错误处理代码：

```
if (error == KIO::ERR_CONNECTION_BROKEN)
     KIO::Scheduler::disconnectSlave(slave);
```

3.8.5 定义一个 ioslave

与服务相似，通过安装一个小的配置文件标识声明新的 ioslave。在 Makefile.am 中安装 ftp.protocol 文件的代码列出如下：

```
protocoldir = $(kde_servicesdir)
protocol_DATA = ftp.protocol
EXTRA_DIST = $(mime_DATA)
```

文件 ftp.protocol 的内容列出如下：

```
[Protocol]    #定义这个 slave 负责的协议
exec=kio_ftp  #定义使用这个 slave 的库名
protocol=ftp
input=none  //这行当前未用到
output=filesystem  //这行当前未用到
listing=Name,Type,Size,Date,Access,Owner,Group,Link,
reading=true
writing=true
makedir=true
```

```
deleting=true
Icon=ftp
```

当这个 slave 支持启动时，kdeinit 启动时按次序装载这个库到其地址空间。这样实际上可以把运行的 slave 作为单独的进程，虽然它是作为库被使用的。这个机制的好处是节约内存并降低运行时的连接时间。

因为 slave 作为共享库被安装，但构成单独的应用程序，其代码框架看起来与正常的共享库插件有一点不同。用来启动 slave 的函数是 kdemain()，它完成一些初始化，并进入一个事件循环。然后等待应用程序使用其请求，例如：

```
extern "C" { int kdemain(int argc, char **argv); }

int kdemain(int argc, char **argv)
{
     KLocale::setMainCatalogue("kdelibs");
     KInstance instance("kio_ftp");
     (void) KGlobal::locale();

     if (argc != 4) {
          fprintf(stderr, "Usage: kio_ftp protocol "
                          "domain-socket1 domain-socket2\n");
          exit(-1);
     }

     FtpSlave slave(argv[2], argv[3]);  //创建slave
     slave.dispatchLoop(); //进入事件循环
     return 0;
}
```

有两个类在应用程序(工作)和 kioslave 之间指定协议，SlaveInterface 类用在应用程序端，SlaveBase 用在 slave 端。slave 应用应该从 SlaveBase 类继承。在 kdevelop 集成开发环境中有生成 kioslave 的向导，指导开发者生成 kioslave 程序框架。

3.9 DCOP

DCOP（Desktop Communication Protocol）的含义是桌面通信协议，其主要目的是为 KDE 应用程序进程之间提供通信。DCOP 是一个客户端到客户端的通信协议，是一种在 socket 上操作的简单的 IPC/RPC 通信机制，支持 unix 域的 socket 或 TCP/IP 的 socket。DCOP 建立在客户端交换协议（Inter Client Exchange，ICE）之上，ICE 是 X11R6 的一部分。这个协议支持消息传送和使用 XML-RPC 到 DCOP "网关"的远端程序调用，并且依赖于 Qt。除此之外，它不依赖于任何库。

DCOP 由个在 KDE 启动时启动的 dcopserver 服务器（后台进程）和多个客户机（支持 DCOP 的应用程序）组成，dcopserver 服务器为客户机通信分发消息。

3.9.1 DCOP 工具

KDE 桌面提供了 DCOP 管理的两个工具，即 dcop 和 kdcop。二者功能是一样的，kdcop 是图形化工具，dcop 是命令行工具，方便于用于脚本程序中。

DCOP 为每个 KDE 应用程序定义了同名或者加上 Iface 的默认操作对象。kdcop 工具以树状视图列出正运行的 DCOP 客户机，展开后可以显示客户机支持的接口函数。双击函数可以调用它并且得到函数执行的结果。如果函数需要参数时，还会弹出对话框允许用户输入。

例如：KDE 桌面环境的一个 E-mail 客户端软件 KMail 有一个接口 checkMail，作用是连接邮件服务器并检查是否有新的邮件到达。用户能够通过如下 dcop 命令进行操作：

```
# dcop kmail default checkMail
```

3.9.2 DCOP 收发数据

模型是简单的，使用 DCOP 的每个应用程序是一个客户端，它们通过一个 DCOP 服务器通信。服务器的功能如同一个交通指挥者，分发消息给到合适的目标。 DCOP 有两种可能的动作，即“send”或“call”. send 是“发送并忘记”消息（这是非阻塞的），call 是“呼叫”（这种方式阻塞等某些数据的返回）。send是异步的，call 是同步的。

所有数据都通过 QDataStream进行序列化，然后被发送。另外有一个与 IDL 类似的编译器(dcopidl 和 dcopidl2cpp）可以生成 Stubs（存根）和 Skeletons（基干）

stub 可以允许把 dcop 注册应用程序作为其方法，即它自己的 dcop slot 的对象。通常，应用程序都有一个 appnameIface.h 文件，其中包含了所有 dcop slot。

（1）建立连接

调用 KApplication::dcopClient()函数会返回一个 DCOPClient 实例的指针，一个应用程序只有一个 DCOPClient 实例。这个函数在第 1 次调用时会创建实例，然后调用这个函数仅返回所在应用程序的 DCOPClient 实例。DCOP 的客户端有惟一的标识，这些标识基于 KApplication::name() 的返回值来确定。

为了激活 DCOP通信，必须使用 DCOPClient::attach()来将信道粘附到 DCOP 服务器上。如果粘附不成功，函数 attach()将返回 FALSE。这时，KApplication 将捕获 dcop信号并显示一个错误消息对话框。

使用 DCOPClient::attach() 连接服务器后，必须注册 appId 以便服务器知道信道的存在；否则通信将是匿名的。注册方法如下：

```
appId = client->registerAs(kapp->name());
```

KUniqueApplication 自动注册其自身到 DCOP，如果使用 KUniqueApplication，则不需要调用 attach() 和 registerAs()。appId 等于 kapp->name()。

（2）发送数据给远程应用程序

可调用 send 或 call 方法发送消息，两个函数有 3 个参数，即应用程序 ID、远程对象和远程函数。发送是异步的（如立即返回），可调用 send或 call 方法。其中远程对象必须被定义为一个对象层次，如顶层对象为 fooObject 有子对象 barObject，应以 fooObject/barObject引用这个对象。函数是全函数描述，如 doIt(int)。

为了实际得到远程客户端的数据，这些数据必须通过 QDataStream 序列化，并使用 QByteArray 操作。

下面是一个客户端发送消息的简单例子。

如果需要调用 doIt，并且非阻塞，则不接收远程调用函数的返回值，但在 RPC 被处理时不会被挂起。DCOPClient::send()返回值表示 DCOP 通信是否成功。

```
QByteArray data;
QDataStream arg(data, IO_WriteOnly);
arg << 5;
if (!client->send("someAppId", "fooObject/barObject", "doIt(int)", data) )
    qDebug("there was some error using DCOP.");
```

如果我们想得到远程调用函数的返回值数据，则必须调用 DCOPClient::call() 以替代 DCOPClient::

send()。返回值将存放在参数 reply 中，call()返回值表示 DCOP 通信是否成功。调用 DCOPClient::call()的样例代码列出如下：

```
QByteArray data, replyData;
QCString replyType;
QDataStream arg(data, IO_WriteOnly);
arg << 5;
if (!client->call("someAppId", "fooObject/barObject", "doIt(int)",
                        data, replyType, replyData))
  qDebug("there was some error using DCOP.");
else {
  QDataStream reply(replyData, IO_ReadOnly);
  if (replyType == "QString") {
      QString result;
      reply >> result;
      print("the result is: %s",result.latin1());
  } else
     qDebug("doIt returned an unexpected type of reply!");
}
```

（3）通过 DCOP 接收数据

为了接收消息，用户的类必须从 DCOPObject 类继承，DCOPObject 提供了纯虚函数 DCOPObject::process()方法。为了处理接收的 DCOP 消息，必须实现这个函数。下面是一个实现样例：

```
bool BarObject::process(const QCString &fun, const QByteArray &data,
                    QCString &replyType, QByteArray &replyData)
{
   if (fun == "doIt(int)") {
        QDataStream arg(data, IO_ReadOnly);
        int i; //用来传入参数
        arg >> i;
        QString result = self->doIt (i);  //调用函数
        QDataStream reply(replyData, IO_WriteOnly);
        reply << result;    //传入函数 doIt 的返回值
        replyType = "QString";
        return true;
   } else {
        qDebug("unknown function call to BarObject::process()");
        return false;
   }
}
```

（4）接收呼叫（Call）并处理

如果应用程序能够处理输入函数，上面的代码是需要的。当应用程序需要完成更复杂的任务时，可能想在 process 函数调用外进行处理，并且以后当结果可用时才发送回结果。

对于这种情况，可以向 DCOPClient 请求一个 transactionId。然后可以从函数 process 返回，并且当结果可用时完成这个任务。同时，应用程序可以从其他的客户端接收 DCOP 函数调用。

函数 process 的样例代码列出如下：

```
bool BarObject::process(const QCString &fun, const QByteArray &data,
                    QCString &, QByteArray &)
{
   if (fun == "doIt(int)") {
        QDataStream arg(data, IO_ReadOnly);
        int i;  //传入参数的变量
        arg >> i;
        QString result = self->doIt(i);  //调用函数

       DCOPClientTransaction *myTransaction;
```

```
        myTransaction = kapp->dcopClient()->beginTransaction();  //请求开始事务

        // 开始处理...
        // 当被动局面时调用 slotProcessingDone
        startProcessing( myTransaction, i);

        return true;
    } else {
        qDebug("unknown function call to BarObject::process()");
        return false;
    }
}

slotProcessingDone(DCOPClientTransaction *myTransaction, const QString &result)
{
     QCString replyType = "QString";
     QByteArray replyData;
     QDataStream reply(replyData, IO_WriteOnly);
     reply << result;  //函数返回结果放入 reply 中
     kapp->dcopClient()->endTransaction( myTransaction, replyType, replyData ); //完成事务
}
```

（5）使用 dcopidl 编译器

dcopidl 使得建立一个 DCOP 服务器变得容易，代替使用 process()方法，可用 dcopidl 创建需要的代码。同样也允许在一个分开单独的头文件中描述自己的类接口。

写一个 IDL 文件与写一个正常的 C++头文件相似，但有一个例外的关键词“ASYNC”，它表示对这个函数的调用将采用异步处理。对于 C++编译器，它扩展到 void。

样例如下：

```
#include <dcopobject.h>

class MyInterface : virtual public DCOPObject
{
  K_DCOP
  k_dcop:

    virtual ASYNC myAsynchronousMethod(QString someParameter) = 0;
    virtual QRect mySynchronousMethod() = 0;
};
```

从上面可以看出， MyInterface类基本上声明一个抽象基类，它从 DCOPObject 类继承。

如果使用标准的 KDE 编译脚本，那么可简单地添加文件（ MyInterface.h）到有源代码目录。然后编辑 Makefile.am 文件，增加“ MyInterface.skel ”到 SOURCES 列表，并且添加 MyInterface.h 到 include HEADERS中。

编译脚本将使用 dcopidl 分析 MyInterface.h 语法，将其转换为一个 MyInterface.kidl 文件中的 XML 描述，然后一个名为 MyInterface_skel.cpp 文件将被自动创建、编译并链接到二进制程序中。

下一步是选择在 MyInterface.h 中描述的接口使用的类，这个类从 MyInterface.h 中定义的接口类 MyInterface 继承，这个选择类的样例如下：

```
class MyClass: public QObject, virtual public MyInterface
{
  Q_OBJECT

  public:
    MyClass();
    ~MyClass();

    ASYNC myAsynchronousMethod(QString someParameter);
```

```
    QRect mySynchronousMethod();
};

MyClass::MyClass()
  : QObject(),
    DCOPObject("MyInterface")
{
    // whatever...
}

void MyClass::myAsynchronousMethod(QString someParameter)
{
    qDebug("myAsyncMethod called with param `" + someParameter + "'");
}
```

显然，这个样例不需要定义作为接口的抽象类，我们可以仅仅直接定义 k_dcop 部分。这样，MyClass类声明变为如下代码：

```
class MyClass: public QObject, virtual public DCOPObject
{
  Q_OBJECT
  K_DCOP

  public:
    MyClass();
    ~MyClass();

  k_dcop:    //仅直接定义k_dcop 部分
    ASYNC myAsynchronousMethod(QString someParameter);
    QRect mySynchronousMethod();
};
```

对于 skeleton，dcopidl2cpp还产生 stub，这使得调用 DCOP 接口变得容易。为了使用 stub，增加 MyInterface.stub 到 Makefile.am 文件的 SOURCES 列表中，这个 stub 类会被 MyInterface_stub 调用。

（6）用户间通信

有时候，可能需要属于不同用户的进程间 DCOP。例如，前端进程使用用户的 ID 运行，后端进程运行作为 root 用户。为了实现这种情况，必须采用两步运行，即两个进程需要与同一个 DCOP 服务器交互通信，以及授权必须被保证。

对于第 1 步，简单地传递服务器对第 2 个进程的地址（在 .DCOPserver 中可找到）。对于授权来说，可使用 ICEAUTHORITY环境变量告诉第 2 个进程在哪里找到授权信息（注意：这将暗示着第 2 个进程能读取授权信息）。这样，如果第 2 个进程运行作为 root 用户，它将可能工作；如果运行作为另一个用户，可以使用 kdesu 进行切换，方法如下：

```
ICEAUTHORITY=~user/.ICEauthority kdesu root -c kcmroot -dcopserver `cat ~user/.DCOPserver`
```

这行命令的含意是在 kdesu 得到 root 口令后，以 root 用户运行 kdmroot 与用户的 dcop 服务器对话。注意：DCOP 通信是非加密的，因此不要用其传递重要信息。

（7）DCOP 协议描述

各种 DCOP协议消息的描述格式及说明如下：

```
#不期望回应的 DCOPSend 消息
data: << fromId << toId << objId << fun << dataSize + data[dataSize]

#能得到一个 DCOPReply、DCOPReplyFailed 或 DCOPReplyWait 回应的 DCOPCall 消息
data: << fromId << toId << objId << fun << dataSize + data[dataSize]

#DCOPReply 是成功回应 DCOPCall 消息
```

```
data: << fromId << toId << replyType << replyDataSize + replyData[replyDataSize]

#DCOPReplyFailed 消息指示 DCOPCall 消息的失败
data: << fromId << toId

#DCOPReplyWait 消息指示一个 DCOPCall 消息被成功处理但回应较晚

data: << fromId << toId << transactionId

#DCOPReplyDelayed 消息是在一个 DCOPReplyWait 消息后对 DCOPCall 消息的成功回应
data: << fromId << toId << transactionId << replyType << replyData
```

DCOPFind 消息类似于 call 消息，但其被送到一个客户端的多个对象。如果一个被调用的对象中的一个函数返回 true 值，则含有对象的 DCOPRef 引用的 DCOPReply 消息将被送回。

（8）保护和重入

当 DCOP 被调用时，为了响应 call，一个 dcop 客户端将监控 dcop 连接。在这个期间，当接收到一个进来的 call 时，正常情况下它将不被处理而排队，直到输出的 call 已被处理。

然而在上面这种情况下，如果进来的 call 是输出 call 的直接或间接结果并且在输出 call 上回应正在等待进来 call 的结果，则将造成死锁。如一个诸如客户端 A 呼叫客户端 B 的环形的 call，同时进来的客户端 B 正在呼叫 A。

为了阻止这种情况下的死锁，DCOP 有一个检测环形 call 的跟踪机制。当检测到一个进来的环形 call 被排队并作为引起死锁的结果，它将立即处理进来的 call 而不是将其排队。这意味着进来的 call 可能在输出 DCOP 的 call 被处理之前被处理，一个应用程序应该能注意这种重入。一个特殊的情况是当一个 DCOP 产生一个对自身的 call，这样的 call 总是直接被处理。

call 跟踪工作通过添加一个 key 到每个输出的 call。当一个客户端接收到一个输入的 call，这个 call 正在等待输出 call 的回应，它将检查输入 call 的 key 是否等于上一次输出 call 的 key。如果 key 等于环形 call，则重入的 call 已被检测到。

如果客户端使用的 key 尚未接收到任何 key，则其将为 0。在这种情况下，服务器将为其送回一个惟一的 key，它们将在后面的处理中用到。如果作为对输入 call 的回应，一个客户端产生一个输出的 call，则其为输出 call 使用输入 call 的 key，代替从服务器接收 key。

值为 1 的 key 有特殊含义，它被诸如 DCOPSend、DCOPReplyFailed 和 DCOP 信号的非呼叫信息使用。

值为 2 的 key 有特殊的含义，它被高优先级的 call 使用。当一个 dcop 客户端在高优先级 call 模式，它将仅处理 key 值为 2 输入 call。

注意：如果客户端 A 和客户端 B 同时彼此互相调用，将可能还有死锁的风险。因为两个 call 将有惟一的 key，并且两个客户端将决定排队输入 call 直到其接收到输出 call 的回应。

（9）DCOP 信号

有时候一个组件需要通过 DCOP 发送“通知”给其他的组件，但不知哪个组件对这些“通知”感兴趣。在这种情况下可使用广播的形式，但这种方法很粗糙，更好的方法是 DCOP 信号。

DCOP 信号与 Qt 信号非常相似，但也有一些差异，一个 DCOP 信号能被连接到一个 DCOP 函数。无论何时发送 DCOP，信号连接的 DCOP 函数被调用。关键词 k_dcop_signals 被用做 DCOP 信号的声明，示例如下：

```
class Example : virtual public DCOPClient
{
   K_DCOP

 k_dcop:
    // 这是一些常规的 DCOP 方法
    ...
```

```
k_dcop_signals:
    // 我们DCOP信号
    void clientDied(pid_t pid);
...
}
```

一个 DCOP 信号从 DCOP 对象/DCOP 客户端联合体（发送者）发送，它能被连接到另个 DCOP 对象/客户端联合体（接收者）的一个函数。

注意：在 Qt 信号的连接与 DCOP 信号的连接之间有两个主要的区别。与 Qt 不同，在 DCOP，一个信号连接能有一个匿名的发送者，并且一个 DCOP 信号连接可以是非 volatile 的。

使用 DCOP，一个对象可未指定发送的 DCOP 对象或 DCOP 客户端而连接一个信号。在这种情况下，来自任何 DCOP 对象和 DCOP 客户端的信号将被分发，从而允许某一事件的指定没有绑定其自身到某一运用这一事件的对象。

另一个 DCOP 特征被称为“非 volatile 连接”。对于 Qt 信号连接来说，当信号的发送者或接收者被删除时，连接被删除。一个 volatile DCOP 信号连接将是同样的，但一个非 volatile DCOP 信号在信号的发送者或接收者被删除时连接将不被删除。一旦一个新的对象用与开始发送对象相同的名字创建，连接将被恢复，二者之间没有区别。在 volatile DCOP 信号连接情况下，信号连接将总被删除。

当发送都还不存在时，接收者能创建一个非 volatile 连接，一个匿名 DCOP 连接应该总是非 volatile 的。

下面的例子说明了无论何时 KLauncher 注意到通过其被启动的应用程序关闭时，它如何发出一个信号：

```
QByteArray params;
QDataStream stream(params, IO_WriteOnly);
stream << pid;
kapp->dcopClient()->emitDCOPSignal("clientDied(pid_t)", params);
```

KDE panel 的任务管理器连接到这个信号，它使用一个非 volatile 匿名连接（不请求被 KLauncher 发送的信号），这个连接列出如下：

```
connectDCOPSignal(0, 0, "clientDied(pid_t)", "clientDied(pid_t)", false);
```

3.10 KDE 应用程序启动其他应用程序的方法

在 KDE 中有多种不同方式从用户的应用程序中启动其他应用程序，下面进行简单说明。

（1）fork + exec

除非在无法使用 KProcess 时，才能使用这种方法。

（2）Kprocess

如果需要启动一个新进程，这个新进程是你的进程的子进程，则使用 KProcess 类，这样你可以得到 stdout/stderr 信息或通过 stdin 输入信息。你不要用这个类启动其他 KDE 应用程序，如果需要从发送文本数据到这个进程或从这个进程接受数据，则最好使用 KProcIO。

（3）KprocIO

与 KProcess 类似，只是 KProcIO 类更容易与创建的进程进行数据交换。

（4）startServiceByDesktopPath

用来启动桌面（KDE/Gnome/X）应用程序或 KDE 服务。应用程序/服务必须有一个 .desktop 文件，它使用 KDEinit 增加启动执行效率并减低内存使用率，作为 KDEinit 可装载模块（KLM）的应用程序才有这些好处。

（5）Krun

打开 documents/applications/shell 命令的方法。Krun 可启动任何应用程序，从二进制到 desktop 文件。它在运行处理程序之前决定一个文件的 mime 类型，并且它还可以启动一个 shell 命令。Krun 是运行 KDE 中另外一个 KDE 应用程序推荐方法。

3.11 国际化和本地化

国际化指应用程序能在任何本地化情况下运行，数据输出（字符串消息）、数据输入（包括输入法）、日期的格式、数字及币种等适合本地化的习惯。国际化简称“i18n”（Internationalization 字符串的前后各取一个字母，共 18 个字符）。

本地化是指国际化应用程序的实际处理适合于特定国家或地区的习惯，它简称为“l10n”（Localization 字符串的前后各取一个字母，共 11 个字符）。

为了支持 i18n 和 l10n，KDE 在 kdelibs/kdecore 目录中提供了 KLocale 类，该类提供了多个函数使代码能本地化。

为了翻译，编程者必须对所有需要显示的字符串使用 i18n()。i18n()使用 char* 作为一个参数，返回一个翻译的 QString 类型字符串。如果没有翻译，则返回原字符串的 QString 类型。

KDE 还提供了 I18N_NOOP()函数标识需要翻译的字符串，这些字符串将被提取并包含进 .po 文件。I18N_NOOP()典型用在 KAboutData 的字符串，因为它在 KApplication 之前被构造，而 i18n()仅在 KApplication 之后使用。

```
QString msgStr = i18n("Creating index file: 1 message done",
                 "Creating index file: %n messages done", num);

QString msg=i18n("Do you want to replace %1 with %2?").arg(oldFile).arg(newFile)
```

开发者必须添加一个消息目标到应用程序的 Makefile.am 中，一个样例消息目标列出如下：

```
messages: rc.cpp
          #$EXTRACTRC 和$EXTRACTATTR 从 xml 文件中提取消息
          $(EXTRACTRC) *.rc */*.rc >> rc.cpp
          $(EXTRACTRC) sidebar/trees/history_module/history_dlg.ui >> rc.cpp
          #提取消息放入到 konqueror.pot 中
          $(XGETTEXT) rc.cpp *.h *.cc *view/*h *view/*cc kedit*/*.cpp \
          -o $(podir)/konqueror.pot
```

（1）字符串翻译过程

脚本 kde-common/makemessages 周期地运行在 KDE 服务器，这个程序从用户应用程序中提取所有的 i18n 消息及在 Makefile.am 中的每个消息目标，提供的消息保存在 l10n 模块下的模板目录的模板（.pot）文件中。

KDE 翻译者将消息进行翻译，并保存在相应语言的消息文件夹中，按模块分开保存，如 konqueror 应用程序的 German 翻译消息保存在 l10n/de/messages/kdebase/konqueror.po 中。

当 l10n 模块被编译时，.po 文件为了快速查找而被编译进一个二进制文件，并以 .mo 文件安装在 $KDEDIR/share/locale/xx/LC_MESSAGES/ 中，其中 xx 表示是语言的 ISO 639 编码的两个字符。

在运行时，i18n()函数使用最初的字符串，在用户的桌面语言对应的目录的消息表中查找字符串。如果未找到，i18n()函数显示最初始的字符串。

在项目的 .desktop 文件中，用户可见的字符串是单独处理的。makemessages 提取字符串，如 .desktop 中的 Name 项和 Comment 项，并把它们放到一个 desktop_mmmm.pot 文件中，其中 mmmm 代表在模板文件夹中的模块名。一旦翻译者翻译了这个文件，makemessages 将翻译的字符串插回到 .desktop 文

件中。

（2）手册

用 docbook 格式书写的手册处理不同于应用程序。翻译的手册保存在每种语言下的 docs 文件夹下的 l10n 模块中，如 German Kate 手册放在 l10n/de/docs/kdebase/kate/文件夹中。在编译时，German Kate 手册安装在 $KDEDIR/share/doc/HTML/de/kate/index.cache.bz2 中。

3.12 编译方法

对于通常的应用程序，开发者写好含有预定义宏的文件 configure.in 和 Makefile.am。由 autoconf 命令根据文件 configure.in 生成 configure，由 automake 命令根据宏文件 Makefile.am 生成 Makefile.in，然后使用 configure 根据据 Makefile.in 来生成用来编译安装程序的 Makefile。

KDE 编译系统使用 admin 目录下 am_edit 程序改进 automake 的性能，方法是 automake 把 Makefile.am 转换为 Makefile.in 文件。然后 am_edit 运行并用 KDE 特有标签的真实含义取代 Makefile.in 文件中所有的 KDE 特有标签，configure 会根据 Makefile.in 文件创建一个 Makefile 文件

KDE 使用脚本程序封装 autoconf 和 automake 命令来完成从 Makefile.am 生成 Makefile 的工作，下面说明 Makefile.am 的编写方法及生成 Makefile 的方法。

1．定义一个应用程序的 Makefile.am

Makefile.am 文件中经常用到的系统变量的使用方法说明如下。

（1）bin：把 kdialog 安装有 KDE 的 bin 目录下。

（2）*_PROGRAMS：编译一个程序。

（3）*_SCRIPTS：一个脚本。

（4）*_SOURCES 源代码列表。

（5）*_LDADD：定义连接哪些库，可以使用 -lfoo ../path/to/thelib.la （仅支持相对路径），或被 kde 编译系统设置的宏 $(LIB_*)。如果库 A 依赖于库 B，则不必定义库 B 但如果 A 的依赖关系改变，确信在 B 之前定义 A。因此通常首先是本地的 .la 库，然后是$(LIB_*)。如果 foo 存在，则然后是-lfoo。

（6）*_LDFLAGS：定义传入到连接器的参数，这个参数可能是 -L 标识。$(all_libraries)含有所有的需要在 Qt 和 KDE 中找到的 -L 标识。对于应用程序来说，应该使用 $(KDE_RPATH)帮助应用程序在不必设置 LD_LIBRARY_PATH 的情况下查找库。

（7）AM_CPPFLAGS：附加编译标识，必须含有 $(all_includes)。并且也包含其他编译时标识，如 -DWITHOUTBUGS 和 -I$(srcdir)/subdir。注意-I 项在$(all_includes)项之前，以确保使用你自己的头文件，而不是一些旧的安装版本。

注意$(srcdir)和“.”（表示编译的目录）总是自动被包含，可能在 KDE 中找到很多 Makefile.am 文件还在使用 INCLUDES。这是其旧的名字，AM_CPPFLAGS 是现在被推荐使用的。

（8）KDE CXXFLAGS：其参数被追加到其他标识后面，这些参数可通过编译框架来设置或取消，如程序出错调试打印设置默认为取消。如果需要这个设置，可通过下面的一行来设置：

```
KDE_CXXFLAGS = $(USE_EXCEPTIONS)
```

（9）METASOURCES = AUTO：使用 KDE 编译系统自动注意 moc 文件。如果所有 .cpp/.cc 文件包括其 .moc 文件或者编译单个二进制/库，则这是推荐的方法。

在同目录下有多个二进制/库的情况下，必须使用下面这样的格式：

```
libsomething_la_METASOURCES = myfile.moc myotherfile.moc
mybinary_METASOURCES = someotherfile.moc
```

示例：一个简单程序的 Makefile.am

```
#bin 意味着将把 kdialog 安装有 KDE 的 bin 目录下，*_PROGRAMS 表示将编译一个程序
bin_PROGRAMS = kdialog
# kdialog_SOURCES 表示应用程序 kdialog 的源文件列表
kdialog_SOURCES = kdialog.cpp widgets.cpp ...
#表示应用程序 kdalog 的连接库
kdialog_LDADD = $(LIB_KIO)
#传入到连接器的参数
kdialog_LDFLAGS = $(all_libraties) $(KDE_RPATH)
#附加编译标识
AM_CPPFLAGS = $(all_includes)
#使用 KDE 编译系统自动注意 moc 文件
METASOURCES = AUTO
```

2. 编写一个共享库的 Makefile.am

当程序代码可被多个应用程序重用时，最好编译为共享库的形式。共享库的 Makefile.am 文件编写与下类似：

```
AM_CPPFLAGS = $(all_includes)
#lib_表示库装在 lib/.目录下，*_LTLIBRARIES 意为 LibTool libraries，表示使用 libtool 处理它，必须使用这个工具处理库
lib_LTLIBRARIES = libkong.la
# *_LIBADD 表示 lib 依赖的库的链表，注意 LDADD 是应用程序用的，而 LIBADD 是 lib 用的
libkonq_la_LIBADD = $(LIB_KPARTS)
# *_LDFLAGS 含有传入连接器的标识，通过对-L 标识使用$(all_libraries)，但还需要库的版本号
libkonq_la_LDFLAGS = $(all_libraries) -version-info 6:0:2 -no-undefined
libkonq_la_SOURCES = popupmenu.cc knewmenu.cpp ...

METASOURCES = AUTO
```

3. 插件/模块的 Makefile.am

所有动态打开的代码（插件、part 组件、kdeinit 模块、applet 及 kioslave）可称为一个“DSO”（动态共享对象）或一个“模块”，所有的模块将被安装在“ KDE 模块”目录中，这个目录通常是 KDEPREFIX/lib/kde3。因此，与共享库的主要区别是模块在 Makefile.am 中应使用与下类似的语句：

```
kde_module_LTLIBRARIES = something.la
```

命名规范和各种类型模块的 _LDFLAGS 的细节请参考 KDE 在线文档。

4. 编写一个静态库的 Makefile.am

如果使用同一选项编译同一源文件到同一目录中的两个共享库或两个应用程序，则可以把源文件列在 _SOURCES 行中。它将仅被编译一次，并且对象可被两个目标使用。然而如果使用同一源文件在一个库和一个程序中，或在不同的目录中共享它，那么不能将这个文件列在两个 _SOURCES 行中。这是因为库和应用程序需要不同的编译选项（-fPIC），并且在不同目录的情况下 automake 不支持其他目录中的源文件。代替的办法是应该把共享的源文件放在其他共享库（已安装的），或在一个静态库（在 Unix 中是 *.a 文件）中，被连接进静态库的目标（共享库或应用程序）实际上是非一体化的目标文件。

```
noinst_LTLIBRARIES = libcommon.la
libcommon_la_SOURCES = dirk.cpp coolo.cpp ...
#不需要 LIBADD or LDFLAGS，如果这个代码需要$(LIBJPEG)等，它能提供帮助，这个静态库的所有用户不必指定它

#可以使用静态库
mylib_la_LIBADD = libcommon.la
myprogram_LDADD = libcommon.la
```

这也是在子目录中编译源文件的方法，不允许使用 automake。如下：

```
foo_SOURCES = subdir/bar.cc
```

必须在 subdir 中定义一个静态库并从顶层目录中使用它。

5. 自动测试用的 Makefile.am

下面是一个自动测试程序的 Makefile.am 样例：

```
METASOURCES = AUTO
#表示是输入 make check 命令行时编译的应用程序
check_PROGRAMS = mytestprog.cpp
#表示是输入 make 命令行时运行的应用程序
TESTS = mytestprog

mytestprog_SOURCES = mytestprog.cpp
mytestprog_LDFLAGS = $all_libraries) $(KDE_RPATH)
mytestprog_LDADD = ../libcommon.la

AM_CPPFLAGES = -I$(srcdir)/..$(all_includes)
```

在应用程序中增加自动测试的一个通用方法是创建一个名为“ tests ”的子目录，在父 Makefile.am 中添加 tests 到 SUBDIRS 行中。然后在新的 tests 目录下创建一个 Makefile.am，使用 create_makefile 工具创建一个新的 Makefile 当输入 make check 时，测试程序将被编译（由于 check_PROGRAMS）并运行（由于 TESTS）。如果测试程序返回一个非 0 错误，将打印错误消息并停止。如果不需要自动运行测试程序，省略 TESTS 行。

注意测试程序需要连接到测试的库，不能连接到 part（组件）、plugin（插件）或 binary（二进制程序）等。

在测试程序中需要命令行参数时，代替 tests，使用 check-local 目标启动测试程序，列出如下：

```
check-local: mytestprog
     ./mytestprog $(srcdir)/datafile
```

6. 安装数据

下面是 Makefile.am 文件中安装头文件的样例语句：

```
include_HEADERS = foo.h bar.h
```

如果类使用了名字空间，如 KParts，那么头文件应被安装为 kparts/foo.h，方法如下：

```
kpartsincludedir = $(includedir)/kparts  # 定义目录
kpartsinclude_HEADERS = foo.h bar.h  #安装头文件到定义目录
```

为了安装数据文件到一个标准目录中，使用 dirname DATA。例如，一个 Type=Service 的 .desktop 文件应进入$(kde_servicesdir)，因此应该如下填写：

```
kde_services_DATA = foo.desktop
```

为了安装数据文件到一个用户目录中，必须首先定义这个目录，然后如下使用它：

```
#安装 bar.desktop 到 KDEPREFIX/share/apps/kmyapp
myappfoodir = $(kde_datadir/kmyapp
myappfoo_DATA = bar.desktop
```

对于 K 菜单入口使用如下语句：

```
xdg_apps_DATA = kmyapp.desktop
```

7. 卸载一个.desktop文件

如果重命名或删除一个.desktop 文件，为了从源代码安装提供平滑的迁移，可能需要卸载旧的文件并在安装时用一个 deleted 文件重写这个文件。

第 1 步添加一个 uninstall.desktop 到源代码中，如果需要，它可在子目录中被共享。这个文件应包括的行如下：

```
[Desktop Entry]
Encoding=UTF-8
Hidden=true
```

然后在 Makefile.am 中加入下面的行：

```
install-data-local: uninstall.desktop
$(mkinstalldirs) $(DESTDIR)/$(kde_datadir)/kmyapp  #只是一个.desktop安装的样例目录
$(INSTALL_DATA) $(srcdir)/uninstall.desktop $(DESTDIR)
$(kde_datadir)/kmyapp/oldfilename.desktop
```

8. 翻译

为了使应用程序可翻译，必须在代码中对显示给用户的英语字符串使用 i18n()，并且在你的顶层 Makefile.am 中定义一个消息目标。

一个脚本运行并查询调用所有的消息目标，最后在 kde-i18n 模块中创建.pot 文件，翻译者能根据这些消息进行翻译后放在.po 文件中。

对于应用程序，运行下面的命令产生 pot 文件：

```
make -f admin/Makefile.common package-messages
```

运行命令 make merge 产生消息融合（新旧.po 文件融合在一起）。重要的是为了处理上下文翻译和复数，必须使用 gettext 工具（可从 ftp.kde.org 下载）。

消息目标应该在含有某一 i18n 的源代码上简单地调用 xgettext，这个源代码不等于_SOURCES，因为_SOURCES 含有不能与 xgettext 一起工作的.ui 和.skel 文件。因此必须用 *.cpp *.h 这样的文件替代（如果有子目录，确信包括子目录）。在简单的情况下，在 Makefile.am 中一个消息目标语句类似如下：

```
messages:
    $(XGETTEXT) *.cpp *.h -o $(podir)/mypotfile.pot
```

注意：.pot 文件名必须是 KDE 中惟一的，因此最好使用应用程序名或模块名作为前缀，如 kofficefilters.pot。

mypotfile 是主 KInstance 的名字（对于一个应用程序来说，它是传递到 KAboutData 中的第 1 个参数；对于组件（part 和 plugin）来说，它是创建的 KInstance 的名字。

如果有.ui（Qt designer）、.rc（XML-UI）或.kcfg 文件，则必须添加类似下面的代码：

```
messages: rc.cpp
```

脚本将在当前目录中从.ui 和.rc 文件创建 rc.cpp。如果在子目录中有.ui、.rc 或 *.kcfg 文件，则必须在 xgettext 调用之前添加类似下面的代码：

```
$(EXTRACTRC) */*.rc >> rc.cpp
```

另一个特殊情况是 tips 文件，即一个装有显示给用户的 tip 的 XML 文件。在这种情况下，添加类似于 下面的代码：

```
messages:
    #使用在 kdebase/ktip 目录下 preparetips 脚本
    perl ./preparetips > tips.cc
    $(XGETTEXT) *.cpp *.h tips.cc -o $(podir)/mypotfile.pit
    rm -f tips.cc
```

最后是编译和安装 .po 文件，在 kde-i18n 中，Makefile.am 文件中含有如下语句：

```
#意味着所有在这个目录下的.po 文件被安装为 language 语言
KDE_LANG = language
POFILES = AUTO
```

如果添加一个分开独立的应用程序，并且希望这个应用程序安装其自己的 .po 文件，则通常将有个 po/ 子目录，并且在 Makefile.am 中将简单地表示为：

```
POFILES = AUTO
```

在这种情况下， .po 文件依赖于其名字 $(PACKAGE).mo 被拷贝，如 de.po 被安装在 de/LC_MESSAGES/$(PACKAGE).mo 并且 fr.po 被安装在 fr/LC_MESSAGES/$(PACKAGE).mo 中。

9. 仅有 Qt 的应用程序

默认情况下，编译系统假定编译 KDE 应用程序，它需要 kdecore（如在编译 Qt Designer 的 .ui 文件时使用 klocale.h）。当仅编译 Qt 应用程序时，使用下面的选项关闭默认的设置：

```
KDE_OPTIONS = qtonly
```

10. 将 Makefile.am 转换为 Makefile

当改变了一个 Makefile.am，并且已有一个从其产生的 Makefile，则可以简单地使用 make，并自动重新产生 Makefile。

如果还没有 Makefile，则可以在顶层目录使用 make -f Makefile.cvs，并重新运行 configure 产生所有的 Makefile,或者在 kdesdk 目录(作为 kdesdk 的一部分被安装)下的 scripts 目录中使用 create_makefile 脚本。使用这个脚本的方法如下：

```
create_makefile subdir
```

在 subdir 父目录中，将从 subdir/Makefile.am 文件创建 subdir/Makefile 文件。另外还有一个脚本 create makefiles 可以产生一个目录树下所有 Makefile.am 文件对应的 Makefile 文件。

3.13 桌面框架

3.13.1 Panel Applets

KDE 的面板（Panel）也称为 “kicker”，它允许称为 “ Applets” 的小应用程序嵌入到面板中。例如，时钟和 kpager（桌面预览 Desktop Overview ）被应用作为 Applets。Applet 是一段用于显示单个小窗口的程序，该窗口位于 KDE 面板或 kicker 中，它通常显示在屏幕的底部。除了这种窗口操作的限制之外，Applet 可以与其他任何程序一样复杂。

Applets 能在内部（如：panel 的进程空间）或外部（如：独立的进程）运行。内部运行时，内存占用小，但不稳定的 Applet 会使整个 Panel 崩溃；外部运行时，将不会发生这种情况，它仅通过 DCOP 连接与 Panel 通信。

从程序员的角度看，内部和外部的 Applet 是透明的。Applet 被用做动态可装载库并且从 libkdeui 库中的 KPanelApplet 类继承。为了通知 Panel 有一个新的 Applet 产生，必须安装一个 .desktop 文件到 $KDE/share/apps/kicker/applet 目录中。Panel 扩展描述文件用类似的文件存于 $ KDEDIR/share/apps/kicker/extensions 目录中。

Panel Applet 从基类 KPanelApplet 继承，提供了一个工厂函数和描述文件的安装。

当插件是共享库并被装入 Panel 的地址空间时，为了安全和稳定性原因，kicker 也通过外部的代理进程 appletproxy 和 extensionproxy 装载插件。装载行为由用户通过 kcontrol 配置，这样可避免因为插件的 bug 造成整个 panel 的崩溃。

我们用简单的“Hello World”Panel Applet 为例说明 Panel Applet 的编写方法，这个例子中的 HelloWorldApplet 类从 KPanelApplet 类继承，它提供了一个工厂函数和一个 .desktop 描述文件。

HelloWorldApplet 声明列出如下：

```
class HelloWorldApplet : public KPanelApplet  //从KPanelApplet类继承
{
    Q_OBJECT
public:
    HelloWorldApplet( const QString& configFile, Type t = Stretch,
                      int actions = 0, QWidget *parent = 0,
                      const char *name = 0 );
    /*重实现了常量虚拟函数 int widthForHeight(int)和 heightForWidth(int)。与 kicker 一起，可以安排 panel 水平放置或垂直放置*/
    //水平放置的 panel，固定宽度，用此函数查询 applet 的高度
    int widthForHeight( int height ) const;
    //垂直放置的 panel，固定高度，用此函数查询 applet 的宽度
    int heightForWidth( int width ) const;
};
```

HelloWorldApplet 类实现列出如下：

```
HelloWorldApplet::HelloWorldApplet( const QString& configFile,
                                    Type type, int actions,
                                    QWidget *parent, const char *name )
  : KPanelApplet( configFile, type, actions, parent, name )
{
    setBackgroundColor( blue );  //设置背景为蓝色
    setFrameStyle( StyledPanel | Sunken );  //设置样式
}

int HelloWorldApplet::widthForHeight( int h ) const
{
    return h;
}

int HelloWorldApplet::heightForWidth( int w ) const
{
    return w;
}

extern "C"
{
    KPanelApplet* init( QWidget *parent, const QString& configFile )
    {
        KGlobal::locale()->insertCatalogue( "helloworldapplet");
        //构造 HelloWorldApplet 类实例，这里的构造函数只传递了默认参数到 KPanelApplet 构造函数
        return new HelloWorldApplet( configFile, KPanelApplet::Normal,
                                     0, parent, "helloworldapplet");
    }
}
```

描述文件 helloworldapplet.desktop 的内容列出如下：

```
[Desktop Entry]
Name = Hello World
Comment = Hello World Applet
#panel 用来定位动态共享库
X-KDE-Library = libhelloworldapplet
X-KDE-UniqueApplet = true  #仅能存在一个实例
```

automake 的输入文件 Makefile.am 的内容列出如下：

```
INCLUDES = $(all_includes)
lib_LTLIBRARIES = libhelloworldapplet.la  #生成库
libhelloworldapplet_la_SOURCES = helloworldapplet.cpp
METASOURCES = AUTO
noinst_HEADERS = helloworldapplet.h
lnkdir = $(kde_datadir)/kicker/applets
lnk_DATA = helloworldapplet.desktop
EXTRA_DIST = $(lnk_DATA)

libhelloworldapplet_la_LDFLAGS = $(all_libraries) -version-info 1:0:0 -module \
                                 -no-undefined

libhelloworldapplet_la_LIBADD = $(LIB_KDEUI)

messages:  #消息目标，提供需要国际化的字符串
     $(XGETTEXT) *.cpp *.h -o $(podir)/helloworldapplet.pot
```

使用下面的命令编译和安装 Applet，方法如下：

```
/configure -prefix=<your-kde-dir>
make
su -c 'make install'
```

现在已经安装 Applet，一个“ Hello World ”条目将显示在 Panel 菜单的 Add-Applet 子菜单中。选择“ Hello World”条目，可以看见蓝色的 Panel Applet 显示在 Panel 中。还可看到一个小灰色的 handle 显示在 Applet 的左边，使用鼠标右键可弹出上下文菜单，也可用鼠标拖动 handle。

这个 Applet 运行后在 kicker 上的显示如图 3-3 所示。



图 3-3 “HelloWorld” Applet 在 kicker 上的显示

3.13.2 控制中心 模 块

KDE 控制中心是一个模块化的应用程序，它是控制中心模块的容器，模块被应用作为动态可装载库。为了写一个控制中心模块，用户的类必须从 libkdeui 库中的 KCModule 类继承，KCModule 类从 QWidget 类继承。基本上，仅仅必须重实现装载和存储配置模块条目的配置方法。为了通知控制中心有关一个新模块，它必须安装一个.desktop 文件在$KDEDIR/applnk/Settings 中。

模块在原理上是一个被改变的简单的窗口部件显示条目，这个模块有很小的接口。控制中心提供所有必要的“粘胶”逻辑和 GUI。为了写一个 config 模块，必须创建一个含有工厂函数的库，如下：

```
#include
typedef KGenericFactoryQWidget> YourKCModuleFactory;
 K_EXPORT_COMPONENT_FACTORY( yourLibName, YourKCModuleFactory );
```

3.14 组件框架

3.14.1 KParts 组件

组件的主要目的是提高代码的重用性，图形组件的框架有 KDE 的 KParts、IBM 及 Apple 的 OpenDoc、Microsoft 的 OLE和 Gnome 的 Bonobo 等。KParts 差不多相当于 ActiveX(即 COM 和 OLE)，它的最大好处是从程序员的角度来看用起来很简单。

个 KDE 组件称为一个“part”，它包括 3 个部分，即窗口部件、与窗口部件一起的功能和这些功能的接口。例如，文本编辑器 KParts 组件由下列元素组成。

（1）多行编辑窗口构件。

（2）附加功能，例如“查找”、“替换”、“拼写检查”及“自动换行设置”等。

（3）用于窗口构件功能的菜单项和工具栏按钮。

KParts 是 KDE part 的框架，基于标准的 KDE/Qt 对象，如 QWidget 和 KTMainWindow。它定义了个非常简单的类集合，即 part、插件、主窗口和 part 管理器。

part 是一个 KDE 组件的名字，为了定义一个新的 part，必须提供窗口部件、用于访问 part 的功能的动作和描述在用户接口中这些动作布局的 XML 文件。

个插件定义了在应用程序用户界面中一些动作，如计算器插件。一个 KParts 主窗口是特殊的 KTMainWindow，其用户界面使用 XML 描述。part 管理器是一个更抽象对象，任务是处理 part 的激活和取消当然，part 管理器仅仅在主窗口嵌入多个 part时才有用。如 Konqueror，它的每个视图是一个 part。KParts 组件框架与应用程序的接口关系如图 3-4 所示。

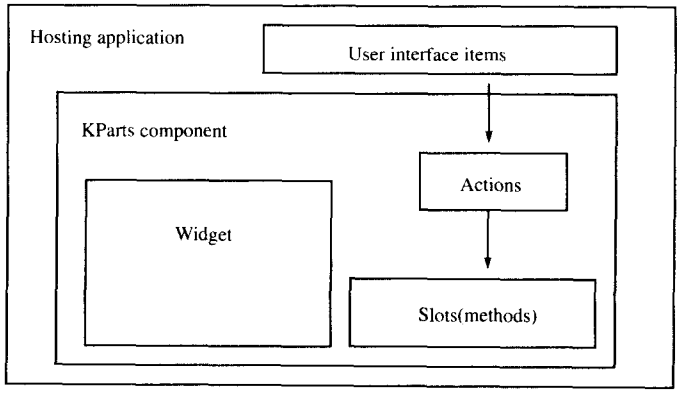


图 3-4 KParts 组件框架与应用程序的接口关系

KDE图形组件框架库设计了窗口部件，这些窗口部件根据动作（菜单条目及工具栏图标等）定义用户接口，见 KDE 在线文档 KParts::Part 类说明。

这个库还提供了使用 part 的应用程序的框架，这些应用程序必须从 KParts::MainWindow 类继承主窗口,并提供 shell GUI。shell GUI 提供了带有 part无关的功能/动作的基本的骨架。KParts::MainWindow 是 KMainWindow 的扩展版本，它提供了将部件的 GUI 合并到主窗口 GUI 中的功能。

KParts 组件被编译为共享库，应用程序使用 KLibLoader 类定位库，打开它并调用初始化函数。这个初始化函数是库的入口点，它从类 KLibFactory 类继承，创建一个工厂。创建部件的工厂代码通常是“样板”代码，它对于所有部件都是相同的。

KDE 3 提供了宏和模板，创建工厂的类似代码列出如下：

```
#include <kparts/genericfactory.h>
// KDE3 的工厂代码
typedef KParts::GenericFactory<KXMLTestPart> KXMLTestFactory;//创建工厂
/*使工厂在外部可用，即在 libkxmltextpart 库中可用，这个宏生成 extern "C" 符号定义，链接为 C 函数，它是库的入口点*/
K_EXPORT_COMPONENT_FACTORY( libkxmltestpart /*库名*/, KXMLTestFactory );
```

3.14.2 系统托盘

一个托盘窗口是一个小的窗口（典型为 24×24 像素点），放在桌面 Panel 的系统托盘上，并且通常显示一个图标或动画图标。图标代表应用程序，与任务栏按钮类似，但占用较少的屏幕面积。例如，图 3-5 所示为在 panel 的系统托盘上显示一个 CD 播放器和一个 Notes 应用程序。



图 3-5 在 panel 的系统托盘上显示一个 CD 播放器和一个 Notes 应用程序

当用点击图标时，应用程序主窗口被显示/升起并被激活。用鼠标右键将得到应用程序特定命令的弹出菜单，其中包含有“ Minimize/Restore”和“ Quit ”菜单项。

在系统托盘中有一个图标，对于后台应用程序来是说一个有用的技术。后台应用程序可能在没有用户交互状况下运行一段时间，但当用户需要时必须立即出现。例如， kppp、kisdn、kscd、kmix 或 knotes 应用程序。使用 kppp 和 kisdn 时，图标提供了网络状态的实时信息。

通过 libkdeui 库中的 KSystemTray，KDE提供了一个非常简单的 API 来从应用程序添加系统托盘条目。非 KDE 应用程序也可根据系统托盘停靠的协议使用系统托盘。

3.14.3 集成 Java

KDE 提供了 kdejava库，使得 KDE应用程序对 Java 小应用程序的支持变得容易。kdejava 库提供了高层次的 API，允许 Java applet 作为一个类似正常的 QWidget 被处理。KDE 的 Java 支持有两部分，即一个纯粹的 Java 服务器提供 Java applet 运行的环境和 KDE 库处理 applet 窗口嵌入应用程序的工作。Java 和 C++代码通过标准的基于简单文本协议的 UNIX 管道来通信，通过 KJAS 服务器可使用任何 Java VM（Java虚拟机 ）。一个简单的集成 Java 的应用程序样例列出如下：

```
#include <kapp.h>
#include <kjavaappletwidget.h>
#include <qstring.h>

int main(int argc, char **argv)
{
 KApplication app( argc, argv );
 QString a,b,c,d,e,f;

 // applet 的名字
 a = "fred";
 // 将运行类的名字
 b = "Lake.class";
 // 基本的 url，也可写作"file:/dos/pcplus/java/lake/Lake.html"
 c = "file:/dos/pcplus/java/lake/";

 // 创建 applet 顶层窗口部件
```

```
    KJavaAppletWidget *applet = new KJavaAppletWidget();
    CHECK_PTR( applet );

    // 下面必须在调用 show()之前设置
    applet->setAppletName( a );
    applet->setAppletClass( b );
    applet->setBaseURL( c );

    // 测试 applet 需要指定一个 image 作为参数
    d = "image";
    e = "arch.jpg";
    applet->setParameter( d, e );

    // 显示窗口部件
    applet->show();

    // 启动事件循环
    app.exec();
}
```

3.15 KDE 协议说明

KDE协议说明如下。

1. 窗口管理器

Extended Window Manager Hints（EWMH） 定义了窗口管理器、应用程序和桌面环境组成部分工具之间的交互的规范，它建立在 ICCCM（Inter-Client Communication Conventions Manual ,客户间通信规范手册）之上。ICCCM 在一个较低的层次上定义了窗口管理器的交互，它不提供现代桌面所期望的许多特征。GNOME 和 KDE 项目最初定义了其自身的对 ICCCM 的扩展。后来，这个规范用一套任何桌面环境能采用的标准 ICCCM 附件代替了 GNOME和 KDE项目定制的扩展。

EWMH 规范描述了下面的问题。

（1）允许客户端改变它们与最大化、阴影、窗口位置固定、桌面及窗口叠加次序等相关的初始状态。

（2）改进窗口管理器的能力，即通过客户端向窗口管理器暗示其窗口类型改变，以及窗口装饰和维护窗口叠加次序。

（3）使 pager 和 taskbar成为分开的客户端，允许其与任何适应的窗口管理器一起工作。

EWMH 规范不覆盖如下内容。

（1）诸如 ICE 或 Corba 这样的其他 IPC 机制。

（2）窗口管理器配置。

（3）窗口管理器文件。

（4）窗口中的窗口 （MDI 多文档界面）。

窗口管理器被认为对窗口管理策略负责，因些无论如何编写客户端，在用户屏幕上应该有一致的行为。

这个规范提供了许多窗口管理器动作的扩展控制，主要是允许 pager、taskbar 和相似的窗口管理器 UI 作为独立的客户端使用。通常客户端除了可能对一个直接的用户请求做出反应外，不应该使用这些。例如，设置配置选项或指定一个 -desk n参数来对应用程序启动窗口最大化进行设置。

当规范说“发送一个消息给根窗口”，可以理解为客户端创建了一个特定内容的 ClientMessage 事件，并使用带有下面参数的 SendEvent 请求发送：

```
destination    root
```

```
propagate       False
event-mask      (SubstructureNotify|SubstructureRedirect)
event           特定的ClientMessage
```

规范中定义了 ClientMessage 消息，如 _NET_SUPPORTED、_NET_CLIENT_LIST 及 _NET_NUMBER_OF_DESKTOPS 等，其中包括根窗口及应用程序窗口属性，以及窗口管理器协议等。

2．会话管理

KDE 支持并使用标准的 X11R6会话管理协议 XSMP，并且支持遗留的 X11R4 和 ICCCM 会话管理协议。

当注销（logout）时，X11R6 会话管理协议为应用程序提供保存其依赖的设置的机会。例如，一个文本编译器将保存装载文件的名字，并在再次注册（login）时重新装载它们。这个协议还支持清除处理和安全注销处理，即使用户还没有决定下次恢复会话。这个协议为应用程序提供了与用户交互的可能性，以免它们在危险时丢失数据，并在必要时取消关机进程。

在 KDE 中，KApplication 和 KMainWindow 隐藏了会话管理的细节。基本上，KApplication 管理了 KConfig 配置对象 sessionConfig()。通过这个对象，应用程序能实现恢复会话特定的数据。

3．KDE 系统托盘停靠协议

对于应用程序来说，一个托盘窗口仅是另一个带有一个特殊属性定义的顶层窗口。这个属性为 _KDE_NET_SYSTEM_TRAY_WINDOW FOR ，类型是 XA_WINDOW 及 format 32。

这个属性纯粹被用来指示窗口管理器应该把这个窗口作为一个托盘窗口处理，属性值可以更多定义托盘主窗口，如托盘窗口代表的窗口。下面的样例用来声明一个托盘窗口：

```
/**协议类型:
Display dpy;
Window trayWin, forWin;
*/
Atom kde_net_wm_system_tray_window_for = XInternAtom( dpy,
              "_KDE_NET_WM_SYSTEM_TRAY_WINDOW_FOR", False);
XChangeProperty( dpy, trayWin, kde_net_wm_system_tray_window_for, XA_WINDOW, 32,
        PropModeReplace, (unsigned char *)&forWin, 1);
```

当应用程序用 XMapWindow() 映射窗口时，如果可能，窗口将被放在系统托盘上，用函数 XUnmapWindow()或 XDestroyWindow()可将窗口从系统托盘上去掉。

4．Drag-And-Drop（拖放）协议

如今，Drag-And-Drop（DND）是各种应用程序的需要，XDND 标准是 X 桌面的 Drag-And-Drop 协议标准，KDE 将其作为首先的协议。 Qt 文档提供了如何编写 drag-and-drop 应用程序的说明。

5．打印协议

KDE 应用程序打印时产生 Postscript Level 1 格式数据输出，输出能被通过管道直接给支持 Postscript 的打印机或被诸如 ghostscript 的 Postscript 解释器使用，当前 Linux 发布版本默认配置上提供了 Postscript 的支持。

3.16 KDE 启动

KDE 启动次序从 startkde 脚本开始。在大多数情况下，一旦用户已被授权，这个脚本从显示管理器（KDM）得到调用。 startkde 脚本中有两行比较重要，列出如下：

```
LD_BIND_NOW=true kdeinit +kcminit +knotify
kwrapper ksmserver $KDEWM
```

第 1 行开始 kdeinit 主进程，用来开始所有其他的 KDE 进程。在 kdeint后的参数是需要启动的附加服务的名字，“+’表示 kdeinit 需要等待直到服务完成。

第 2 行请求 kdeinit 启动 ksmserver 会话管理器进程，会话管理器决定了会话的生命期。当这个进程退出时，用户被注销。

所有的 KDE 后台服务是用户特定的，与系统后台进程不同，它们不在用户间被共享。 KDE 后台服务对于每个用户是独立的，而且对于每个 Xserver display 也是惟一的。

3.16.1 startkde 脚本分析

系统启动时运行 startx 脚本，该脚本已在第 1 章中论述，startx 脚本调用 startkde 脚本启动 KDE 桌面系统。

startkde 脚本在/usr/bin/startkde 中建立 KDE运行环境、创建菜单、启动 kwin 管口管理器并装载字体库等。 startkde脚本分析如下：

```
#!/bin/sh
#
#  默认 KDE 启动脚本 ( KDE-3.1.3 )
#  Modified for Red Hat Linux
#

unset BLOCK_SIZE # breaks parsing of df output
shopt -u -o noclobber # 允许使用 '>' 对文件的覆盖写

 # 如果环境没有，建立用户环境
 # 检查/tmp 和 "$HOME"， 并检查写权限
 .. 省略检查部分

 #建立配置目录
 [ -d "$HOME"/.kde/share/config ] || mkdir -p "$HOME"/.kde/share/config

 #从/usr/share/desktop-create-kmenu 目录下，读取各顶层菜单的配置，建立顶层菜单
 if [ -x /usr/bin/desktop-create-kmenu ] ; then
   /usr/bin/desktop-create-kmenu >&/dev/null
 fi

 if [ ! -d "$HOME"/.kde -a -d /etc/skel/.kde ]; then
   ( cd /etc/skel && tar cpf - .kde ) | ( cd "$HOME" && tar xpf - )
 fi

 if [ ! -e "$HOME"/.kde/share/config/emaildefaults ]; then
   # '>' 表示输出到右边的文件中
   /usr/bin/mailsettings >"$HOME"/.kde/share/config/emaildefaults
 fi

 if [ ! -e "$HOME"/.kde/share/config/kmailrc ]; then
   #将下面 EOF 之前的配置写入到 kmailrc 文件中
   cat >"$HOME"/.kde/share/config/kmailrc <<EOF
 [General]
 transports=1

 [Transport 1]
 host=/usr/sbin/sendmail
 name=Sendmail
 type=sendmail
 EOF
 fi
```

```
if [ ! -e "$HOME"/.kde/share/config/kcmlaptoprc -a -e /proc/apm ]  ; then
  # 如果是笔记本，尝试配置它
  if ! grep -q "?" /proc/apm ; then
    #写入配置，让电池状态通知信息显示在笔记本上
    cat >"$HOME"/.kde/share/config/kcmlaptoprc <<EOF
[BatteryDefault]
Enable=true
EOF
  fi
fi

# 当 X Server死了无反应时，我们将从 xinit 得到 HUP 信息。我们必须忽略它，因为我们还需要做一些清除工作
trap 'echo GOT SIGHUP' HUP

# 启动次序
#kdeinit 被用来清除派生的子进程，这样改善了内存使用并减少了启动时间
#
# 首先 kdeinit 启动 dcopserver 和 klauncher，然后启动 kded，kded 负责更新 sycoca 数据库。当一个更新的数据库出现时，它进入到后台并继续启动。最后 kdeinit 启动 kcminit，kcminit 执行用户设置的设备的初始化。启动 ksmserver，启动次序如下。
#   (1) kwin 窗口管理器
#   (2) $KDEDIR/share/autostart      中设置的程序，如 kdesktop, kicker 等
#   (3) 会话（session）的其他部分

# 用户的个人 KDE 目录通常是 ~/.kde,但这个设备可被 KDEHOME设置重载
kdehome=$HOME/.kde
#在 kdehome 字符串中，用!$HOME代替 '~' 字符
test -n "$KDEHOME" && kdehome=`echo "$KDEHOME"|sed "s,^\~/,$HOME/,"`

#激 活 kde 字体目录
#有 4 个目录可能被用来支持 KDE 的字体，两个系统目录属于 administrator，两个用户目录，用来添加用户自己的字符。 override 版本字体优先于非 override 版本的字体。例如，如果在 override 目录中有一个字体，它将优先于其他目录被使用。优先次序列出如下：
# user override、system override、X、user及 system
#
# X是在这个脚本运行前被设置的最被字体库

usr_odir=$HOME/.fonts/kde-override  # 字体库 user override
usr_fdir=$HOME/.fonts           #字体库 user
if test -n "$KDEDIRS"; then
  kdedirs_first=`echo "$KDEDIRS"|sed -e 's/:.*//'`  #   用空格代替 '..' 或 '*'
  sys_odir=$kdedirs_first/share/fonts/override  # 字体库 system override
  sys_fdir=$kdedirs_first/share/fonts         # 字体库 system
else
  sys_odir=$KDEDIR/share/fonts/override   # 字体库 system override
  sys_fdir=$KDEDIR/share/fonts         #     字体库 system
fi

# 我们在应用的字体目录运行 mkfontdir（如果有权限）拾取任何将被安装的新字体。如果 mkfontdir 运行失败，我们还添加 user 目录到字体路径中，它们可能简单地被 administrator 只读

test -d "$sys_odir" && xset +fp "$sys_odir"
test -d "$usr_odir" && (mkfontdir "$usr_odir" ; xset +fp "$usr_odir")
test -d "$usr_fdir" && (mkfontdir "$usr_fdir" ; xset fp+ "$usr_fdir")
test -d "$sys_fdir" && xset fp+ "$sys_fdir"

#
#添加任何用户安装的字体目录到 X 字体路径
kde_fontsdir=$kdehome/share/fonts
kde_fontpaths=$kde_fontsdir/fontpaths

if test -r "$kde_fontpaths" ; then
   savifs=$IFS
   IFS=
   for fpath in `grep -v '^[ ]*#' < "$kde_fontpaths"` ; do
      if test -s "$fpath"/fonts.dir ; then
```

```
            xset fp+ "$fpath"
        fi
    done
    IFS=$savifs
fi

# 请求 X11 并建立其字体链表
xset fp rehash

#如果 user 已经覆盖写字体，光标字体可能不同了，不移去它。而设置一个 left 光标代替标准 X11 的“ X ”光标，直到所有的内容显示在屏幕上。
#设置 Red Hat 的背景默认为蓝色
test "$XDM_MANAGED" || bkg="-solid #5477A0"
xsetroot -cursor_name left_ptr $bkg
test -f /etc/X11/Xresources && xrdb -merge /etc/X11/Xresources

#
# Get Ghostscript to look into user's KDE fonts dir for additional Fontmap
if test -n "$GS_LIB" ; then
    GS_LIB=$kde_fontsdir:$GS_LIB
    export GS_LIB
else
    GS_LIB=$kde_fontsdir
    export GS_LIB
fi

#连接 “tmp “资源到 /tmp 里的目录，创建目录/tmp/kde-$USER，并连接$ KDEHOME/tmp-$HOSTNAME到它上面
lnusertemp tmp >/dev/null

# 连接 "socket" 资源到/tmp里的目录，创建目录/tmp/ksocket-$USER，并连接$ KDEHOME/socket-$HOSTNAME 到它上面
lnusertemp socket >/dev/null

# 在 dcop sockets 被以前的 session 剩下的情况下，清除它
dcopserver_shutdown

echo 'startkde: Starting up...' 1>&2

#闪烁屏幕和进度指示
ksplash

#我们设置 LD_BIND_NOW来改进kdeinit 的效率。在装载应用程序前， kdeinit没设置这个变量
LD_BIND_NOW=true kdeinit +kcminit +knotify
if test $? -ne 0; then
  # 启动失败
  echo 'startkde: Could not start kdeinit. Check your installation.' 1>&2
  xmessage -geometry 500x100 "Could not start kdeinit. Check your installation."
fi

#最后，把 session控制交给 session管理器。如果设置了 KDEWM 环境变量，那么，它将代替 kwin 作为 KDE 的窗口管理器；如果没有设置 KDEWM环境变量， ksmserver 将确保 kwin被启动。
#kwrapper 被用来减少启动时间和内存的使用量，它不返回诸如 ksmserver 的 exit 这样有用的错误代码。我们仅检查 255，它意味着 ksmserver 进程可能未被启动而导致的任何问题。如 ksmserver 初始化失败，将无法被检测。
test -n "$KDEWM" && KDEWM="--windowmanager $KDEWM"
kwrapper ksmserver $KDEWM
if test $? -eq 255; then
  # 启动错误
  echo 'startkde: Could not start ksmserver. Check your installation.' 1>&2
  xmessage -geometry 500x100 "Could not start ksmserver. Check your installation."
fi

echo 'startkde: Shutting down...' 1>&2

# 清除工作
kdeinit_shutdown
dcopserver_shutdown
```

```
artsshell -q terminate

echo 'startkde: Running shutdown scripts...' 1>&2

# 运行在$KDEDIRS/shutdown 目录下发现的脚本
for prefix in `kde-config --path exe| sed -e 's^bin/^shutdown/^g;s^:^ ^'`
do
  for file in $prefix/*
  do
    if test -f $file
    then
      $file
    fi
  done
done

echo 'startkde: Done.' 1>&2
```

3.16.2 startkde 中的应用程序说明

当运行 startkde 脚本时，该脚本先后使用了 xset、xsetroot、xrdb、ksplash、kded、kdeinit、kcminit、knotify、ksmserver、KLauncher 命令或应用程序。下面分别说明这些命令的作用。

（1）xset、xsetroot和 xrdb

xset 设置 X 窗口的属性。

xsetroot 设置 X根窗口的参数。

xrdb 是 X 服务器资源数据库实际工具，它用来在 screen 0 屏幕根窗口上得到或设置 RESOURCE_MANAGER 属性内容。通常从 X 启动文件运行这个程序，大多数 X 客户端使用 RESOURCE_MANAGER 和 SCREEN RESOURCES 属性得到用户给应用程序的颜色、字体等设置的优化值。

（2）ksplash、klauncher 和 dcopserver

ksplash 显示 KDE 启动进度的 ksplash 画面。

klauncher 是一个后台进程，它负责 KDE 内的服务激活，并且与 kdeinit 主进程密切连接操作启动新的进程。KDE 应用程序通过 DCOP 与 klauncher 通信来启动新的应用程序或服务。

dcopserver 是一个后台进程，它为所有的 KDE 应用程序提供了进程间通信（DCOP）功能。DCOP 功能通过 dcop 命令行或 kdcop 工具可被访问，DCOP 对所有的 KDE 应用程序来说是基本的。

（3）kcminit 和 knotify

kcminit 在启动期间执行初始化服务，初始化服务通过 X-KDE-Init 行定义在应用程序或服务的 .desktop文件中，它基于用户的指定配置初始化硬件。 kcminit –list 可用来显示所有初始化服务，并且 kcminit <service>可用来明确地执行单个服务。当查询启动问题时，这可能是有用的。

knotify 的主要任务是把声音提醒传给声音服务器，并提供可替代的提醒方法。

（4）kdeinit

当 kdeinit 第 1 次启动时，将启动 dcopserver、klauncher 和 kded，以及任何在 startkde 脚本中 kdeinit 的命令行中指定的程序。正常指定的是 kcminit 和 knotify。

kdeinit 是一个进程启动器，在某种程度上类似于启动 UNIX 的 init 启动器，它通过 fork 创建子进程并接着装载一个含有“main(...)”函数的动态库。 kdeinit 可以用来启动所有其他 KDE 应用程序，包括正常的二进制程序文件和 kdeinit 可装载模块（kdeinit loadable module，KLM）

kdeinit 连接一个标准 KDE 应用程序需要的所有库，使用这种技术，启动一个应用程序快得多，因为应用程序仅需要连接它本身；不足是用 kdeinit 启动的程序，用 top 或 ps 命令显示时出现的都是 kdeinit 进程。例如，使用 top –c 或 ps aux 命令显示如下：

```
> ps aux
waba    23184  0.2  2.1 23428 11124 ?      S    21:41   0:00 kdeinit: Running...
waba    23187  0.1  2.1 23200 11124 ?      S    21:41   0:00 kdeinit: dcopserver --nosid
waba    23189  0.2  2.4 25136 12496 ?      S    21:41   0:00 kdeinit: klauncher
waba    23192  0.7  2.8 25596 14772 ?      S    21:41   0:00 kdeinit: kded
waba    23203  0.8  3.4 31516 17892 ?      S    21:41   0:00 kdeinit: knotify
```

其中，kdeinit: Running…表示进程是 kdeinit，其他进程是作为 KLM 启动的程序。

通过 kdeinit 启动的应用程序的进程名是“kdeinit”，通过改变“ ps ”命令显示的程序名可以在一定程序上纠正这个问题。然而诸如“ killall ”这样的应用程序将仅能把“ kdeinit”作为进程名，为了改进这种情况，kdeinit 启动的应用程序应该在 kdesdk/scripts 中使用“kdekillall”.

下面说明程序应该如何修改来适合于使用 kdeinit 启动。

kdeinit 启动应用程序时，实际上将应用程序作为动态共享库，即其一部分进行运行。为了适应这种变化，使用 kdeinit 时，应用程序的源代码不需要任何变化，仅仅需要修改 Makefile.am。如果一个正常的程序的 Makefile.am类似如下：

```
bin_PROGRAMS = kicker
kicker_LDADD = $(top_builddir)/libkonq/libkonq.la
kicker_LDFLAGS = $(all_libraries) $(KDE_RPATH)
```

则需要修改下面行以生成 kdeinit 可用的库版本：

```
lib_LTLIBRARIES = kicker.la
libkicker_la_LIBADD = $(top_builddir)/libkonq/libkonq.la
libkicker_la_LDFLAGS = $(all_libraries) $(KDE_RPATH) -module
```

（5）kded

kded 是一个通用的 KDE 后台进程，是 KDE Daemon 的缩写。它有能力装载各种服务模块，并且在后台运行这些模块,每个 kded 服务模块被一个安装在$KDEDIR/share/services/kded/目录下的.desktop 文件描述。在控制中心(the Control Center）的服务管理器可以监控服务模块的状态，并取消某些服务。

这些服务模块组成多个小任务，这些小任务中的一些是内建的，其他的则是由命令启动的小任务。

内建的任务说明如下。

当检测到新软件时，检查新安装的软件并更新 ksycoca。由 kded 启动的应用程序 kbuildsycoca 更新 ksycoca 当 kded 首次被启动时，它总是运行 kbuildsycoca ，以确保 ksycoca 被更新。

检查新安装的更新文件，应用程序能安装*.upd 更新文件。当新版本应用程序使用不同的配置文件格式安装，这些*.upd 文件用来更新用户的配置文件，配置文件的更新工作由 kconf_update 进行。当 kded检测到一个新的更新文件时，启动 kconf_update 当 kded 首次被启动时，总是运行 kconf_update，确保不丢失任何更新文件。kconf_update 跟踪配置文件 kconf_updaterc 中哪些更新文件已被处理，每次仅执行一次更新。

改变一个正运行系统的主机名是不好的，它通常仅发生在不合理地配置 dial-up 连接时。尽管很少，kded 将检查主机名的改变。如果检测到，则修改 KDE 环境和 X-server 来允许后续的合理操作，执行 kdontchangethehostname做实际的变化。

通过命令装载的任务说明如下。

无论何时，一个应用程序尝试对 kded 模块产生 DCOP 呼叫。kded 支持用命令装载模块，从而可能对核心管理任务有用。

kded 模块的一个例子是打印模块。当一个应用程序打印一个文件且打印时，打印模块将监视打印工作，从而允许在提供打印命令给打印机后可以关闭应用程序。当一个打印问题发生，如纸用完时，打印模块将确保通知用户。

一个 kded 模块应该安装一个.desktop 文件，该文件应该安装在 ${kde_servicesdir}/kded 中，方法如下：

```
ServicesTypes=KDEDModule
```

如果提供如下行，将在 KDE 启动时安装一个 kded 模块：

```
X-KDE-Kded-autoload=true
```

（6）ksmserver

在 startkde 脚本中，启动 ksmserver 后，启动应用程序的次序如下。

- kwin 窗口管理器。
- $KDEDIR/share/autostart 中设置的程序，如 kdesktop, kicker 等。
- 会话（session）的其他部分。

ksmserver 是 KDE 的会话服务器，在启动时自动启动（auto-start）应用程序，并且从以前的会话中恢复应用程序。

在$KDEDIR/share/autostart 目录中的.desktop 文件指示了可自动启动的应用程序。自动启动的一个应用程序是否含有附加自动启动条件依赖于在 .desktop 文件中的 X-KDE-autostart-condition 条目配置值。例如：ktip.desktop 含有下面一行：

```
X-KDE-autostart-condition=ktiprc:TipOfDay:RunOnStart:true
```

这行意味着需要检查 ktiprc 配置文件中 [TipOfDay] 节下的 RunOnStart 值。如果未发现这个条目，则假定为 true，即 ktip 是默认下自动启动的应用程序之一。

KDE 会话管理器恢复以前的会话，一个会话包含一个应用程序集合及所保存的反映应用程序状态的特定信息。会话保存在 ksmserverrc 配置文件中，其中包括应用程序特定状态信息的引用。应用程序特定状态信息被保存在 $KDEHOME/share/config/session 中。例如，ksmserverrc 含有下面的信息：

```
restartCommand1=kwin,-session,110a0a000200010676068910000001936000O_1083342151_608625
restartCommand2=konsole,-session,10378ac0700001083342047000002777690000_1083342151_552722
```

kwin 和 konsole 应用程序特定状态信息分别在下面两个文件中：

```
$KDEHOME/share/config/session/kwin_110a0a000200010676068910000001936000O_1083342151_608625
$KDEHOME/share/config/session/konsole_10378ac0700001083342047000002777690000_1083342151_552722
```

kwin 的状态信息包含有会话中所有其他应用程序的窗口位置。

3.16.3 桌面组件

桌面的 3 个主要组件是窗口管理器 KWin、桌面本身 KDesktop 和面板 Kicker，下面分别说明其功能。

1. 窗口管理器 KWin

窗口管理器总是作为会话由会话管理器启动，它通过虚拟桌面负责管理所有的窗口，包括窗口的初始放置、最小化、最大化、升起、降低，以及窗口之间的焦点切换等。

2. 桌面 KDesktop

KDesktop 通过 $KDEDIR/share/autostart/kdesktop.desktop 由会话管理器自动启动，KDesktop 负责设置墙纸、激活屏幕保护、处理桌面上的图标及运行命令（Alt+F2）对话框。

在启动时，KDesktop 还将打开/运行在 Autostart 路径中发现的所有文档和应用程序， Autostart 路径在控制中心(the Control Center) 的 Paths 小节配置。默认的 Autostart 路径为$KDEHOME/Autostart。

注意这个 Autostart 路径不同于会话管理器使用的$KDEDIR/share/autostart 目录。还需要注意当$KDEDIR/share/autostart必须包含.desktop 文件时，$KDEHOME/Autostart 可能包含各种文件。这些文件将被打开，如同其被用户用鼠标点击一样。

KDesktop 的主要任务之一是提供桌面图标，这些图标主要反映在 Desktop 路径中发现的文件。Desktop路径在控制中心的 Path 小节中被配置，默认的 Desktop 路径为 $HOME/Desktop，这个文件还在 Desktop 路径中。KDesktop 将显示在$KDEDIR/share/apps/kdesktop/Desktop 中的任何文件，后者是为所有用户提供定制桌面图标的方便方法。

KDesktop运行后显示的窗口如图 3-6 所示。

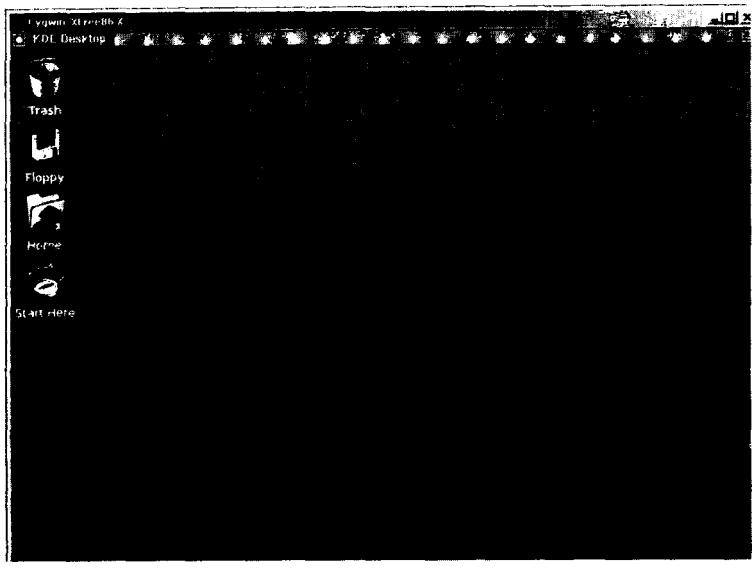


图 3-6 Kdesktop运行后显示的窗口

KDesktop应用程序对应的 kdebase-3.4.0/kdesktop 目录下的文件功能说明如表 3-5 所示。

表 3-5 kdebase-3.4.0/kdesktop 目录下的文件功能说明

文件名（*表示通配符）	功能说明
main.*	main()函数
desktop.*	桌面图标的图标窗口
krootwm.*	鼠标的右键和中键弹出菜单，Mac 菜单栏
bg*.*	背景着色器/管理器
icons.*	旧代码不再使用
KDesktopIface*	KDesktop 的 DCOP 接口，被 kfmclient 使用
lock*	屏幕保护器/锁

屏幕保护与 xscreensaver 的工作方式相似，例如，一个驱动器（KDesktop）提供了定时、加锁和消隐功能，但没有实际上的绘制。一个分开独立的程序提供了在窗口中绘制的能力，这个窗口用来绘制实际想像的图形。 xscreensaver 将与 KDE 屏幕保护引擎一起工作。

屏幕保护通过DCOP 控制，说明如下。

（1）KScreensaverIface::configure() ：从 desktoprc 中重装载屏幕保护配置。

（2）KScreensaverIface::lock()　：立即锁住屏幕。
（3）KScreensaverIface::save()：立即屏幕保护。
（4）KScreensaverIface::isEnabled()：如果屏幕保护被激活，返回 true。
（5）KScreensaverIface::isBlanked()　：如果屏幕当前是消隐状态，返回 true。

3. 面板 Kicker

面板通过$KDEDIR/share/autostart/panel.desktop 被会话管理器自动启动。

3.17　kicker 应用程序

kicker 是 KDE 应用程序启动器面板，除了在 K 菜单中可以启动应用程序外，kicker 还可运行 applets，如 pager、任务栏或时钟及扩展（如子面板）等。

kicker 启动后生成的 kicker 界面如图 3-7 所示。

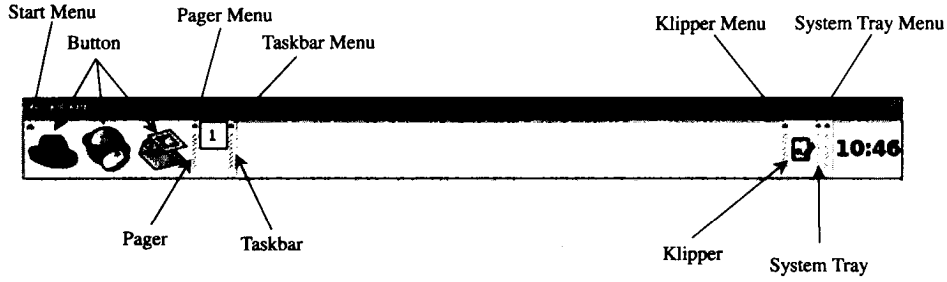


图 3-7　kicker 启动后生成的 kicker 界面、

applet 是运行在 kicker 内部的小应用程序，几乎所有带有启动按钮应用程序都是 applet。任务栏 applet 显示正在运行应用程序的按钮。 pager applet 显示桌面的预览，并允许用鼠标左键点击选择需要使用的桌面。时钟 applet 是嵌入在 kicker 上的显示时间的小时钟。Quick Launcher Applet 是一个缩减的应用程序启动器，系统托盘被用来停靠一些特殊应用程序，如 Klopper 或 KTeaTimer。可以通过 K 菜单中的 Configure Panel 子菜单或上下文菜单增加或删除面板扩展，面板扩展中可放置扩展任务栏等。

kicker 顶层目录内容的说明如表 3-6 所示。

表 3-6　kicker 顶层目录内容的说明

目录名	内容说明
applets/	基本 applet，其中每个类均从 kdelibs/kdeui 中的类 KPanelApplet 继承
buttons/	可在 kicker 中点击的所有按钮，包括 KMenu、quickbrowsers 和应用程序按钮
core/	kicker 核心代码，包括 main()和把每个事件粘合在一起的各种容器与集合类
data/	图标、按钮标题、背景和 KMenu 侧的图像
extensions/	扩展、aka panel，其中每个类从 KPanelExtension 类继承，包括外部任务栏，kasbar、windowmaker doc、applet bar、孩子 panel 和常见的 sidebar
menuext/	类似按钮或菜单，是提供给 panel 的按钮和出现在 Kmenu 中菜单的插件。它们通常创建在 buttons/中，弹出菜单用按钮时会遇到
proxy/	包括 applet 和扩展用的小应用程序，它们允许在进程外运行，用来保护 kicker 安装 applet 或扩展可能出现崩溃的情况。如果它们在内部被装载，则将 kicker 与其放在一起

续表

目录名	内容说明
share/	一些在 core 目录下没有，但对于 kicker 来说是基本的类
ui/	菜单和对话框，菜单的文件名中含有“_mnu”。kicker 的控制面板在 kdebase/kcontrol/kicker 中
taskbar/	实际的任务栏窗口部件，包括在任务栏 applet（applets/taskbar）和任务栏扩展（extensions/taskbar）中
taskmanager/	代表任务的类和任务的集合，由 taskbar/下的任务使用

3.17.1 kicker 应用程序的类继承关系

在 kicker 应用程序中，kicker 类从 KUniqueApplication 类继承，如图 3-8 所示。它完成 kicker 应用程序的配置、发送运行消息给 ksplash 用来显示 kicker 运行进度，作为 X 窗口客户端发送 XEvent 事件并设置资源等。

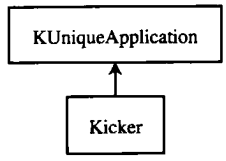


图 3-8 kicker 类的继承关系

kicker 应用程序的窗口中各个窗口部件类的分布图如图 3-9所示。

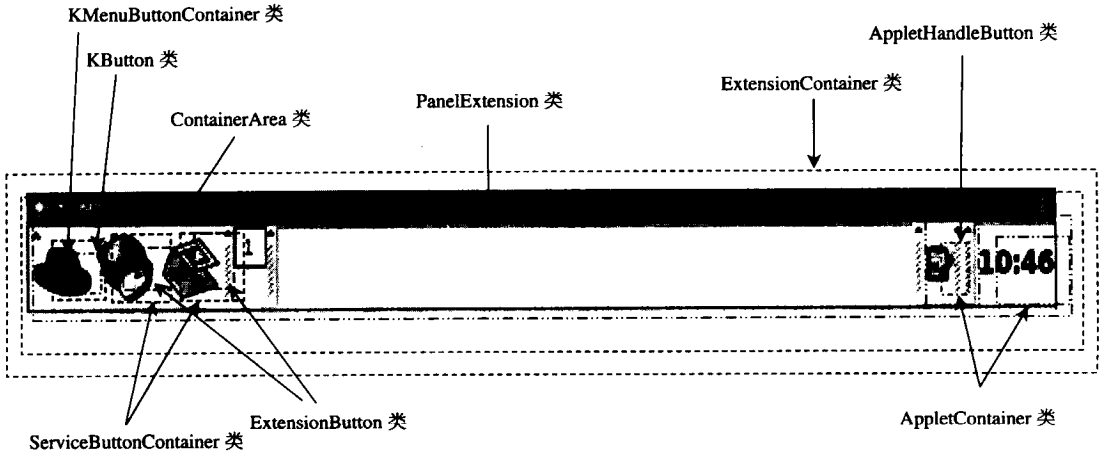


图 3-9 kicker 应用程序的窗口中各个窗口部件类的分布

在 kicker 应用程序的各个类中，ExtensionManager 类提供对 Panel 扩展容器类属性的管理，PluginManager 类提供了对 applet 插件及 Panel 扩展容器类的接口管理，这样可做便于扩展。

下面分别说明构成 kicker 应用程序的各个类。

（1）ExtensionManager 类

ExtensionManager 类从 QObject 类继承，如图 3-10 所示。使用全局链表 ExtensionList 管理 ExtensionContainer 对象，即可以管理多个 Panel，默认只有一个主 Panel。这个链表定义如下：

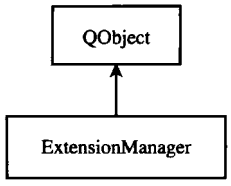


图 3-10 ExtensionManager 类的继承关系

```
typedef QValueList<ExtensionContainer*> ExtensionList;
```

（2）PluginManager类

PluginManager类从QObject类继承，它提供了创建applet窗口容器AppletContainer对象、扩展容器ExtensionContainer对象的方法函数，并将AppletContainer和AppletContainer对象作为插件来管理。此外，该类还提供了装载applet、装载扩展的方法函数，用AppletInfo对象指针链表管理applet。AppletInfo类描述了applet的信息，它将对应Applet的.desktop文件的内容读入到AppletInfo对象中保存。

（3）ExtensionContainer类

ExtensionContainer类从QFrame类继承，如图3-11所示。它是扩展的框架，扩展的各种窗口部件放在这个框架中。一个ExtensionContainer类代表了一个Panel框架，该类提供了设置或得到扩展容器的几何尺寸、位置及panel ID等属性方法，还提供了事件过滤及窗口事件处理等方法。创建Panel框架的方法包括从扩展代理中创建或者构造PanelExtension类对象，具体使用哪种方法，由PluginManager类提供的接口函数loadExtension来判断。

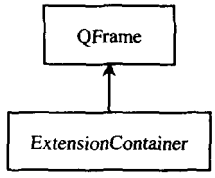


图3-11 ExtensionContainer类的继承关系

（4）PanelExtension类

PanelExtension类从DCOPObject类和KPanelExtension类继承，如图3-12所示。它是具体的Panel的框架并添加了对DCOP的支持。它提供了Panel大小及位置的设置、增加按钮和Applet的方法，以及事件过滤。

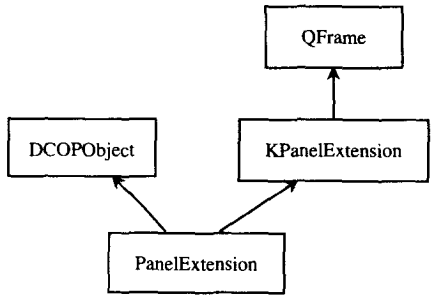


图3-12 PanelExtension类的继承关系

（5）ContainerArea类

ContainerArea类从Panner类继承，如图3-13所示。它实际上是一个视图类，用来容纳各种窗口部件，并且提供了视图滚动及鼠标事件处理等方法。

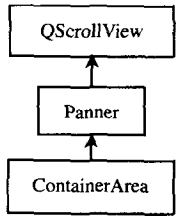


图3-13 ContainerArea类的继承关系

（6）KMenuButtonContainer 类

KMenuButtonContainer 类从 ButtonContainer 类继承，在构造函数中创建 KButton 对象。KButton 类从 PanelPopupButton 类继承，是 K 主菜单的按钮。

（7）DesktopButtonContainer 类

DesktopButtonContainer 类从 ButtonContainer 类继承，它在构造函数中创建了 DesktopButton 对象。DesktopButton 类从 PanelButton 类继承，是显示桌面的按钮。

（8）ServiceButtonContainer 类

ServiceButtonContainer 类从 ButtonContainer 类继承，如图 3-14 所示。它在构造函数中创建 ServiceButton 对象，ServiceButton 类从 PanelButton 类继承，它是应用程序服务的按钮。该类从应用程序的 .desktop 文件中读出配置，设置按钮图标及功能。

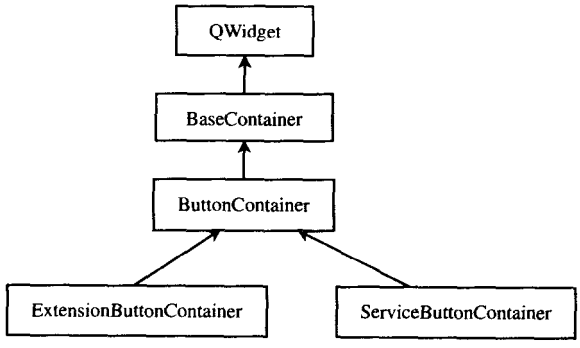


图 3-14 ExtensionButtonContainer 类和 ServiceButtonContainer 类的继承关系

（9）AppletContainer 类

AppletContainer 类从 BaseContainer 类继承，如图 3-15 所示。它在构造函数中创建 AppletHandle 对象。AppletHandle 类构造函数中创建代表 applet 的按钮对象 AppletHandleButton，并根据插件的 .desktop 文件读出配置，在按钮上设置对应插件的图标。

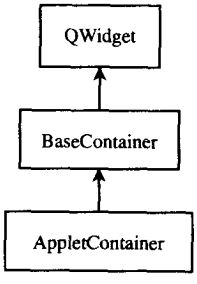


图 3-15 AppletContainer 类的继承关系

3.17.2 桌面配置文件

桌面配置文件描述了应用程序或插件或服务在桌面上的显示信息等，下面列出了对应 .desktop 文件的功能说明。

1. 插件的桌面配置文件

插件的桌面配置文件说明了 applet 插件显示的名字、图标及描述，还说明了插件使用的库和属性等。例如，clock 插件的桌面配置文件 clockapplet.desktop（在/usr/share/apps/kicker/applets 中）列出如下：

```
[Desktop Entry]
Encoding=UTF-8
Name=Clock    #显示的插件名
Name[af]=Horlosie
Name[az]=Saat
......
Comment[zu]=I-applet yewindi lemininingwane yewashi eliqhubekayo noma elezinombolo.  #显示的描述

X-KDE-Library=clock_panelapplet    #插件对应的库文件
X-KDE-UniqueApplet=false      #可存在多个实例
```

2. 应用程序的桌面配置文件

应用程序的桌面配置文件说明了应用程序显示的图标及名字，还提供了应用程序的文件名、帮助文件名、安装版本及安装的菜单位置等信息。例如， kpager 应用程序的桌面配置文件 kde-kpager.desktop（在 /usr/share/applications 中）列出如下：

```
{Desktop Entry]
Encoding=UTF-8
Name=KPager  #显示的应用程序名
Name[af]=Kpager
......
Icon=kpager    #显示的图标
Type=Application  #应用程序类型
Exec=kpager    #应用程序名
DocPath=kpager/index.html  #帮助文件
Terminal=0
X-KDE-StartupNotify=false
X-DCOP-ServiceType=Unique
X-Desktop-File-Install-Version=0.3  #安装版本
Categories=Application;Utility; X-Red-Hat-Base;  #在菜单中的安装位置
OnlyShowIn=KDE;
```

3. 服务桌面配置文件

服务是将功能与具体的应用程序或插件进行分离，即一个服务可以选择具体的实现程序或插件。服务桌面配置文件说明了服务显示的名字及图标，还提供了服务使用的应用程序及帮助文件等信息。例如，服务 khelpcenter 的桌面配置文件（在 /usr/share/services 中）列出如下：

```
[Desktop Entry]
Encoding=UTF-8
Name=KHelpCenter  #显示的服务名
Name[af]=Khulpsentrum
......
Icon=khelpcenter  #显示的图标
DocPath=khelpcenter/index.html  #帮助文件名
Type=Service  #类型是服务
Terminal=0
Exec=khelpcenter %u  #打开服务的应用程序名
SwallowExec=
SwallowTitle=
BinaryPattern=
X-KDE-StartupNotify=true
```

3.17.3 kicker 窗口的建立过程

kicker 应用程序的代码较多，但只要理解类之间继承关系、窗口部件类对应在桌面窗口上的位置，以及桌面窗口的操作次序即可看明白代码。 kicker 应用程序的窗口组织与一般应用程序类似，即由 QFrame 的继承类、QScrollView 的继承类和按钮等小窗口部件类组成。图 3-16 所示为 kicker 应用程序

启动时 kicke r 窗口的建立过程。

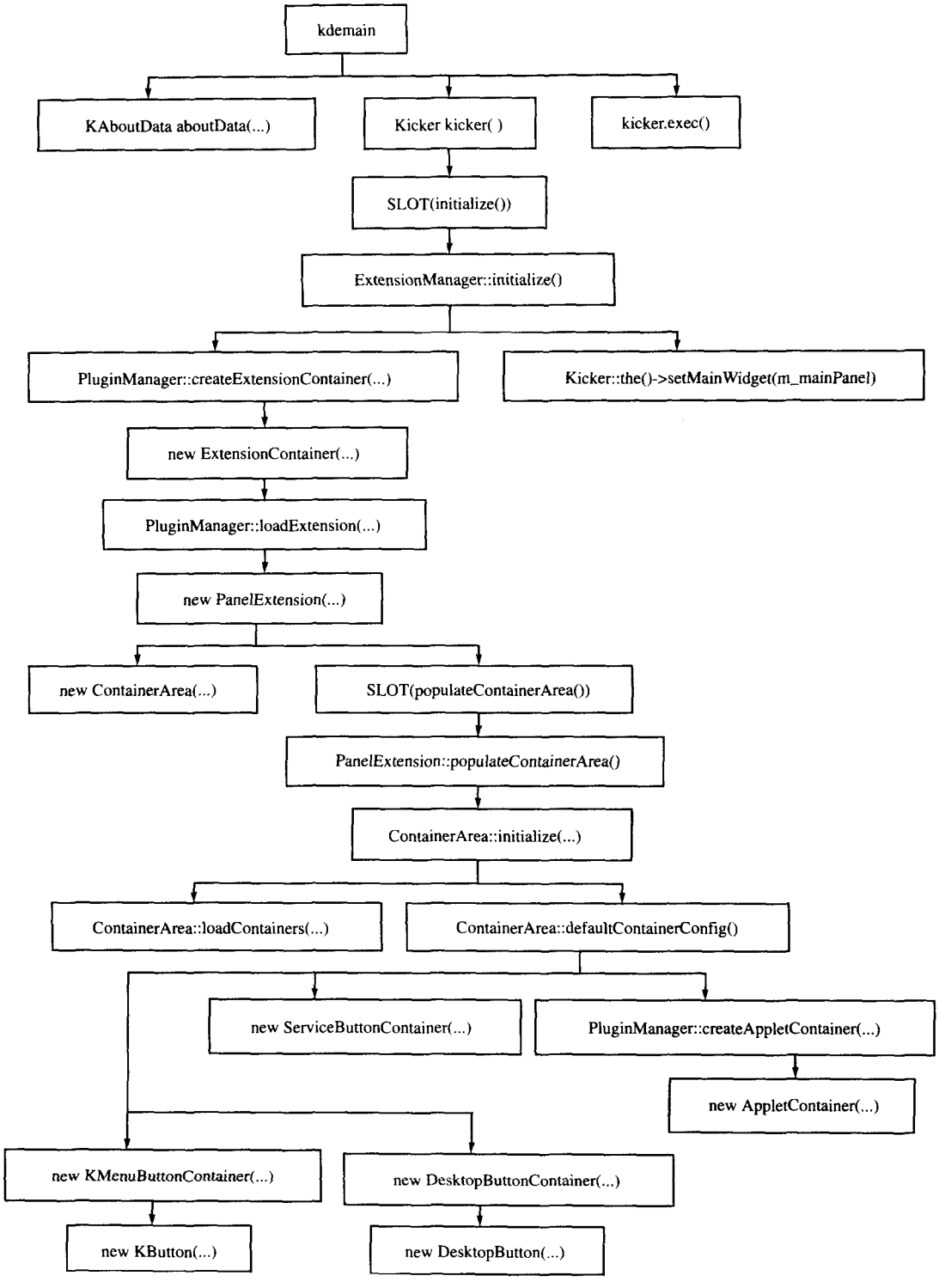


图 3-16　kicker应用程序启动时窗口的建立过程

下面对照图 3-16 分析 kicker 应用程序启动过程中的部分代码。

（1）kdemain 函数

kicker 应用程序因为使用 kdemain 作为 main 函数，所以程序可以以模块方式快速启动。 kdeinit 应用程序用 C 语言编写，因此 kdemain 之前需加上 extern "C"前缀。

函数 kdemain（在 kdebase-3.4.0/kicker/core/main.cpp 中）列出如下：

```
extern "C" KDE_EXPORT int kdemain( int argc, char ** argv )
{
    //省略了多头显示设置
    ...
    QCString appname;
    if (kicker_screen_number == 0)
        appname = "kicker";
    else
        appname.sprintf("kicker-screen-%d", kicker_screen_number);

    //省略了 KaboutData 类应用程序信息设置
    ...
    if (!Kicker::start()) {
        kdError() << "kicker is already running!" << endl;
        return 0;
    }

    if (signal(SIGTERM, sighandler) == SIG_IGN)
        signal(SIGTERM, SIG_IGN);
    if (signal(SIGINT, sighandler) == SIG_IGN)
        signal(SIGINT, SIG_IGN);
    if (signal(SIGHUP, sighandler) == SIG_IGN)
        signal(SIGHUP, SIG_IGN);

    Kicker kicker; //构造 kicker
    return kicker.exec(); //进行事件循环
}
```

（2）Kicker类

Kicker 类是 KUniqueApplication 类的子类，它对 kicker 应用程序进行整体控制。通过静态的函数 Kicker::the()可得到 Kicker 类的静态实例，kicker 启动时构造 Kicker 类实例。

Kicker 类构造函数首先确保其资源被添加到标准目录下，其中包括 tile、背景和各种插件目录，然后注册全局的键盘加速键(accel)。当注册“Popup Launch Menu”accel 时，它引用 MenuManager 创建 KMenu 对象。当安装的.desktop 文件变化时，MenuManager 创建用于更新 KMenu 的设施。

下一步，创建 ExtensionManager (EM)，EM 装载并管理所有 KPanelExtension 子类或 Panel。如果没有扩展被装载，那么 Kicker 用默认设置创建默认 Panel。默认设置基于模板文件。

最后，Kicker 检查是否选择了菜单栏面板，并且是否在屏幕的顶层建立一个 Panel，它为这个 Panel 使用 kickermenubarrc 文件。如果这个 Panel 不存在，则创建它。私有的扩展可能装载各种 applet，还有其他使用 PluginManager (PM)和 EM 的扩展。

Kicker 类的构造函数列出如下：

```
Kicker* Kicker::the() { return static_cast<Kicker*>(kapp); }
Kicker::Kicker()
    : KUniqueApplication(),
      keys(0),
      m_kwinModule(0),
      m_configDialog(0),
      m_canAddContainers(true)
{
    if (KCrash::crashHandler() == 0 )
    {
        // 这意味着我们很可能已崩溃过一次，过 2 分钟后重新设置 crash handler
```

```
        QTimer::singleShot(120000, this, SLOT(setCrashHandler()));
    }
    else
    {
        // 设置crash handler
        setCrashHandler();
    }

    //初始化配置对象kickerrc
    KickerSettings::instance(instanceName() + "rc");

    // 如果配置模块被标识为不可变的，设置config为仅读，即kicker配置不可变
    if (isImmutable() && kapp->authorizeControlModules(Kicker::configModules(true)).isEmpty())
    {
        config()->setReadOnly(true);
        config()->reparseConfiguration();
    }

    dcopClient()->setDefaultObject("Panel");
    //让ksplash显示kicker正在运行
    dcopClient()->send( "ksplash", "", "upAndRunning(QString)", KCmdLineArgs::appName());

#ifdef Q_WS_X11  //发送X事件给X服务器，启动KSplash启动进度显示
    XEvent e;
    e.xclient.type = ClientMessage;
    e.xclient.message_type = XInternAtom( qt_xdisplay(), "_KDE_SPLASH_PROGRESS", False );
    e.xclient.display = qt_xdisplay();
    e.xclient.window = qt_xrootwin();
    e.xclient.format = 8;
    strcpy( e.xclient.data.b, "kicker" );
    XSendEvent( qt_xdisplay(), qt_xrootwin(), False, SubstructureNotifyMask, &e );
#endif

    disableSessionManagement();  //取消会话管理
    //增加资源到标准目录下
    QString dataPathBase = KStandardDirs::kde_default("data").append("kicker/");
    KGlobal::dirs()->addResourceType("mini", dataPathBase + "pics/mini");
    KGlobal::dirs()->addResourceType("icon", dataPathBase + "pics");
    KGlobal::dirs()->addResourceType("applets", dataPathBase + "applets");
    KGlobal::dirs()->addResourceType("tiles", dataPathBase + "tiles");
    KGlobal::dirs()->addResourceType("extensions", dataPathBase +  "extensions");

    KImageIO::registerFormats();

    KGlobal::iconLoader()->addExtraDesktopThemes();

    KGlobal::locale()->insertCatalogue("libkonq");
    KGlobal::locale()->insertCatalogue("libdmctl");

    //初始化加速键
    keys = new KGlobalAccel( this );
#define KICKER_ALL_BINDINGS
#include "kickerbindings.cpp"
    keys->readSettings();
    keys->updateConnections();

    //建立全局设置
    configure();

    connect(this, SIGNAL(settingsChanged(int)), SLOT(slotSettingsChanged(int)));
    connect(this, SIGNAL(kdisplayPaletteChanged()), SLOT(paletteChanged()));

#if (QT_VERSION-0 >= 0x030200) // XRANDR support
    connect(desktop(), SIGNAL(resized(int)), SLOT(slotDesktopResized()));
#endif
```

```
    //panels, aka 扩展
    QTimer::singleShot(0, ExtensionManager::the(), SLOT(initialize()));
}
```

（3）创建按钮及 applet 插件

函数 defaultContainerConfig()描述了在默认配置下创建按钮及 applet 的过程，创建的方法都是从.desktop 文件中读出桌面配置并将配置中的图标及显示名字放在窗口部件上，然后将按钮或 applet 图标与相应的应用程序连接起来。

函数 defaultContainerConfig() 列出如下：

```
void ContainerArea::defaultContainerConfig()
{
    //FIXME: make this use a file template so it isn't hardcoded anymore
    ContainerList containers;
    //创建 K 菜单按钮及显示桌面按钮，将对象添加到链表中
    containers.append(new KMenuButtonContainer(_opMenu, m_contents));
    containers.append(new DesktopButtonContainer(_opMenu, m_contents));

    int dsize;
    if (orientation() == Horizontal)
    {
        dsize = width();
    }
    else
    {
        dsize = height();
    }

    dsize -= 560;
    QStringList buttons;

    //获取按钮的桌面配置文件名
    QFile f(locate("data", "kicker/default-apps"));
    if (f.open(IO_ReadOnly))
    {
        QTextStream is(&f);

        while (!is.eof())
            buttons << is.readLine();

        f.close();
    }
    else
    {
        buttons << "kde-Home.desktop"
                << "kde-konqbrowser.desktop"
                << "kde-Kontact.desktop"
                << "kde-kword.desktop"
                << "kde-Help.desktop";
    }

    //创建按钮窗口部件
    for (QStringList::ConstIterator it = buttons.begin(); it != buttons.end(); ++it)
    {
        BaseContainer *button;
        KService::Ptr service = KService::serviceByStorageId(*it);
        if (!service)
        {
            // 查找特殊的按钮
            QString s = locate("appdata", *it);
            if (s.isEmpty()) continue;
```

```
            QString itExt = (*it).section('/', 1);
            button = new ExtensionButtonContainer(itExt, _opMenu, m_contents);
        }
        else
        {
            button = new ServiceButtonContainer(service, _opMenu, m_contents);
        }

        if (button->isValid())
        {
            containers.append(button);
        }
        else
        {
            delete button;
        }
    }

    PluginManager* manager = PluginManager::the();
    AppletContainer *a = 0;

    // pager applet
    a = manager->createAppletContainer(
        "minipagerapplet.desktop",
        true,
        QString::null,
        _opMenu,
        m_contents);
    containers.append(a);

    // taskbar applet
    a = manager->createAppletContainer(
        "taskbarapplet.desktop",
        true,
        QString::null,
        _opMenu,
        m_contents);
    containers.append(a);

    // system tray applet
    a = manager->createAppletContainer(
        "systemtrayapplet.desktop",
        true,
        QString::null,
        _opMenu,
        m_contents );
    a->setFreeSpace(1);
    containers.append(a);

    // clock applet
    a = manager->createAppletContainer(
        "clockapplet.desktop",
        true,
        QString::null,
        _opMenu,
        m_contents );
    a->setFreeSpace(1);
    containers.append(a);

    for(ContainerIterator it(containers); it.current(); ++it)
    {
        addContainer(it.current());
    }

    saveContainerConfig();
}
```

3.17.4 K 菜单

在 kicker 应用程序中，kicker 使用 MenuManager 类来管理菜单。MenuManager 类是其他应用程序创建菜单的工厂，同时也是这些菜单的拥有者。

MenuManager 类从类 QObject 和类 DCOPObject 继承，它使用 clientmenus 对象指针链表管理 KickerClientMenu 对象。类 KickerClientMenu 处理 PanelKButton 使用的客户端应用程序菜单，管理顶层 K 按钮菜单，并且使用 m kbuttons 链表管理 PanelPopupButton 对象。

MenuManager 类声明中 clientmenus 和 m kbuttons 的声明列出如下：

```
protected:
   PanelKMenu* m_kmenu;
   QPtrList<KickerClientMenu> clientmenus;

private:
   KButtonList m_kbuttons;
```

K 菜单窗口部件主要由类 KButton 和类 PanelKMenu 组成，这两个类说明如下。

类 KButton 是 K 菜单按钮，其继承关系如图 3-17 所示。它设置并初始化 K 菜单。

类 PanelKMenu 是 K 菜单，其继承关系如图 3-18 所示。在继承 PanelServiceMenu 类的情况下，还包含主菜单特有的锁屏、运行命令行、Logout 菜单栏及功能实现函数。PanelServiceMenu 类从 KPanelMenu 类继承，它增加或删除用户新增加的菜单条目，连接各种菜单事件。KPanelMenu 类从 KPopupMenu 类继承，它得到或设置与菜单相关的配置文件路径。

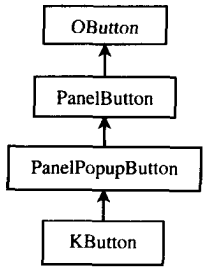


图 3-17 KButton 类的继承关系

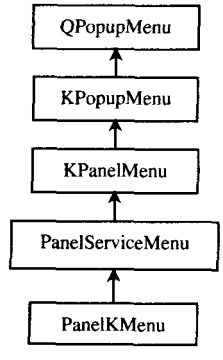


图 3-18 PanelKMenu 类的继承关系

MIME 类型的使用方法是一个文档或代表文档的图标与一个 MIME 类型对应，一个 MIME 类型软个或多个应用程序对应这样，一个文档可以找到一个或多个应用程序打开它。当有多个应用程序可以打开这个文档时，使用优先级最高的应用程序。

K 菜单创建并运行后的显示如图 3-19 所示。

KPanelMenu 类是为 K 菜单或 Panel 动态装载的菜单条目，它创建菜单条目。这些菜单条目被动态地加入到 K 菜单中，或作为一个正常按钮到 Panel 中。动态菜单放在共享库中，共享库在运行时由 kicker 装载。

必须从 KPanelMenu 类继承类来创建一个菜单，并实现虚函数 initialize() 和 slotExec()。开发者还必须在库中提供一个工厂对象，这个工厂仅被用来构造菜单对象。

最后，开发者还必须提供一个.desktop 文件描述自己的动态菜单。相关的条目是 Name、Comment、Icon 和 X-KDE-Library（存有库名，库名不包括扩展名），这个.desktop 文件必须安装在 $ KDEDIR/share/

apps/kicker/menuext/ 目录下。

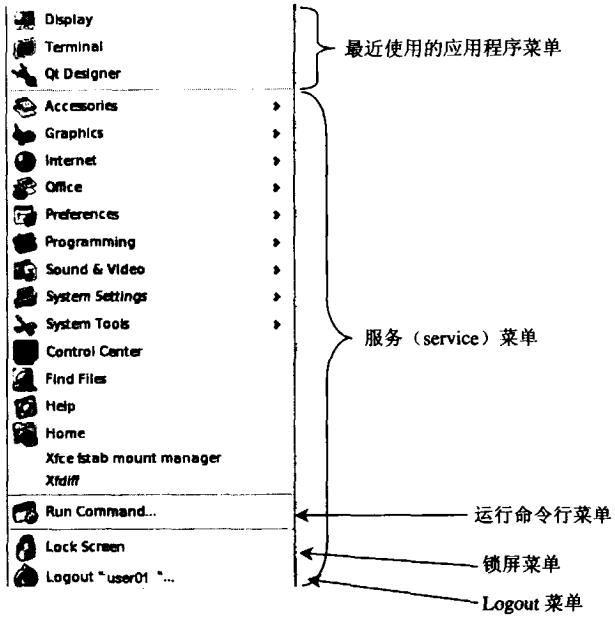


图 3-19　K菜单创建并运行的显示

kicker 应用程序调用 KPanelMenu 类的方法 initialize()创建菜单，这个函数的调用层次如图 3-20 所示。

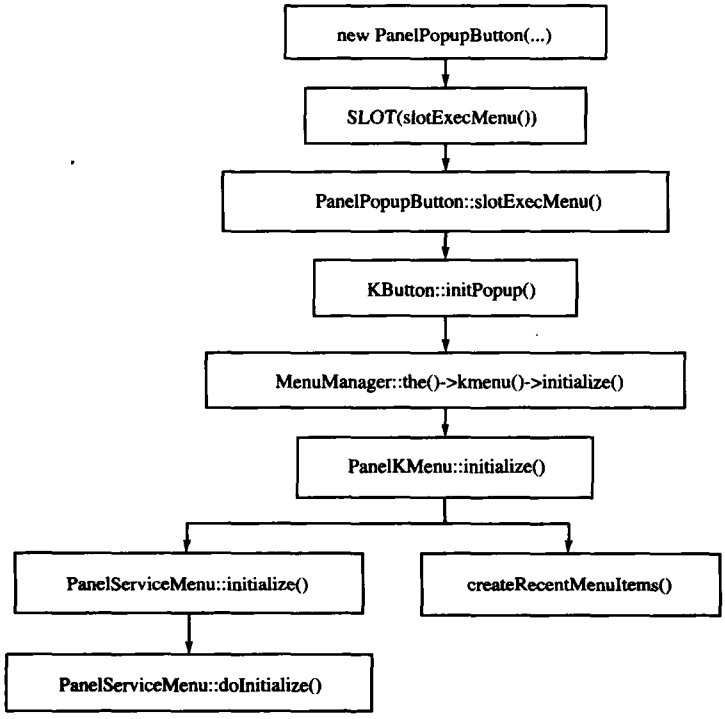


图 3-20　函数 PanelKMenu::initialize()的调用层次

在创建 KButton 按钮时，KButton 类的父类 PanelPopupButton 构造函数将鼠标按键事件与

slotExecMenu()连接起来， PanelPopupButton 构造函数列出如下：

```
PanelPopupButton::PanelPopupButton(QWidget *parent, const char *name)
  : PanelButton(parent, name)
{
   m_pressedDuringPopup = false;
   m_popup = 0;
   setDrawArrow(true);
   connect(this, SIGNAL(pressed()), SLOT(slotExecMenu()));
}
```

槽函数 slotExecMenu 初始化并显示菜单，列出如下：

```
void PanelPopupButton::slotExecMenu()
{
   if (KickerTip::tippingEnabled())
   {
      KickerTip::the()->untipFor(this);
   }

   if (m_popup != 0)
   {
      m_pressedDuringPopup = false;
      KickerTip::enableTipping(false);
      kapp->syncX();  //同步 X
      kapp->processEvents();   //事件处理
      initPopup();             //初始化菜单
      m_popup->adjustSize();  //调整菜单尺寸，以便正确地显示菜单
      //在屏幕上合适的位置及方向运行显示菜单
      m_popup->exec(KickerLib::popupPosition(popupDirection(), m_popup, this));
      setDown(false);
      KickerTip::enableTipping(true);  //激活 tip 帮助条
   }
}

void KButton::initPopup()
{
   MenuManager::the()->kmenu()->initialize();
}
```

函数 initialize()初始化 K 菜单，并加入各种菜单条目，列出如下：

```
void PanelKMenu::initialize()
{
   /*从最近使用过应用程序链表中得到应用程序名对应的 .desktop 文件，并构造成 kservice 对象，将应用程序的标题插入到主菜单上部的最近使用条目中。
   //更新最近使用应用程序菜单条目
   updateRecent();

   if (initialized())  //如果已初始化，则返回
   {
      return;
   }

   if (loadSidePixmap())  //装入侧边图标
   {
      // 断开并重连接
      disconnect(kapp, SIGNAL(kdisplayPaletteChanged()),
                 this, SLOT(paletteChanged()));
      connect(kapp, SIGNAL(kdisplayPaletteChanged()),
              this, SLOT(paletteChanged()));
   }
   else
```

```
    {
        sidePixmap = sideTilePixmap = QPixmap();
    }

    // 加入服务菜单，即各种应用程序菜单条目
    PanelServiceMenu::initialize();

    if (KickerSettings::showMenuTitles())
    {
        int id;
        id = insertItem(new PopupMenuTitle(i18n("All Applications"), font()), -1 /* id */, 0);
        setItemEnabled( id, false );
        id = insertItem(new PopupMenuTitle(i18n("Actions"), font()), -1 /* id */, -1);
        setItemEnabled( id, false );
    }

    // 创建最近使用应用程序菜单条目
    createRecentMenuItems();

    bool need_separator = false;

    // 插入书签
    if (KickerSettings::useBookmarks() && kapp->authorizeKAction("bookmarks"))
    {
        //每次需要创建新的弹出菜单，这个菜单被 subMenus.clear()删除
        KPopupMenu * bookmarkParent = new KPopupMenu( this, "bookmarks" );
        if(!bookmarkOwner)
            bookmarkOwner = new KBookmarkOwner;
        delete bookmarkMenu; //不能再使用旧的菜单，因为它已被删除
        bookmarkMenu = new KBookmarkMenu( KonqBookmarkManager::self(), bookmarkOwner, bookmarkParent,
actionCollection, true, false );

        if (KickerSettings::menuEntryHeight() > -1)
        {
            insertItem(SmallIconSet("bookmark",
                        KickerSettings::menuEntryHeight()),
                    i18n("Bookmarks"),
                    bookmarkParent );
        }
        else
        {
            insertItem( i18n("Bookmarks"), bookmarkParent);
        }

        subMenus.append(bookmarkParent);
        need_separator = true;
    }

    //插入 quickbrowser 菜单
    if (KickerSettings::useBrowser())
    {
        PanelQuickBrowser *browserMnu = new PanelQuickBrowser(this);
        browserMnu->initialize();

        if (KickerSettings::menuEntryHeight() > -1)
        {
            insertItem(SmallIconSet("kdisknav",
                            KickerSettings::menuEntryHeight()),
                    i18n("Quick Browser"),
                    BaseContainer::reduceMenu(browserMnu));
        }
        else
        {
            insertItem(i18n("Quick Browser"),
                    BaseContainer::reduceMenu(browserMnu));
```

```
    }

    subMenus.append(browserMnu);
    need_separator = true;
}

/*插入动态菜单，根据/usr/share/apps/kicker/menuext 目录下的各个.desktop 文件插入动态菜单*/
QStringList menu_ext = KickerSettings::menuExtensions();
if (!menu_ext.isEmpty())
{
    for (QStringList::ConstIterator it=menu_ext.begin(); it!=menu_ext.end(); ++it)
    {
        //根据/usr/share/apps/kicker/menuext 目录下的各个.desktop 文件读入菜单信息
        MenuInfo info(*it);
        if (!info.isValid())
          continue;

        KPanelMenu *menu = info.load();  //根据菜单信息创建菜单
        if (menu)
        {
            if (KickerSettings::menuEntryHeight() > -1)
            {
                insertItem(SmallIconSet(info.icon(),
                                        KickerSettings::menuEntryHeight()),
                           info.name(),
                           menu);
            }
            else
            {
                insertItem(info.name(), menu);
            }
            subMenus.append(menu);
            need_separator = true;
        }
    }
}

if (need_separator)
    insertSeparator();

//如果有客户端菜单，则插入这个菜单
if (clients.count() > 0) {
    QIntDictIterator<KickerClientMenu> it(clients);
    while (it){
        if (it.current()->text.at(0) != '.')
            insertItem(
                it.current()->icon,
                it.current()->text,
                it.current(),
                it.currentKey()
                );
        ++it;
    }
    insertSeparator();
}

// 插入 run command 菜单栏
if (kapp->authorize("run_command"))
{
    if (KickerSettings::menuEntryHeight() > -1)
    {
        insertItem(SmallIconSet("run", KickerSettings::menuEntryHeight()),
                   i18n("Run Command..."),
                   this,
                   SLOT( slotRunCommand()));
```

```
    }
    else
    {
        insertItem(i18n("Run Command..."),
                this,
                SLOT( slotRunCommand()));
    }
    insertSeparator();
}
//插入用户切换菜单
if (DM().isSwitchable())
{
    sessionsMenu = new QPopupMenu( this );

    if (KickerSettings::menuEntryHeight() > -1)
    {
        insertItem(/*SmallIconSet("", KickerSettings::menuEntryHeight())),*/
                i18n("Switch User"),
                sessionsMenu);
    }
    else
    {
        insertItem(i18n("Switch User"), sessionsMenu);
    }

    connect( sessionsMenu, SIGNAL(aboutToShow()), SLOT(slotPopulateSessions()) );
    connect( sessionsMenu, SIGNAL(activated(int)), SLOT(slotSessionActivated(int)) );
}

//如果用户配置了 ksmserver，插入会话相关操作菜单栏
KConfig ksmserver("ksmserverrc", false, false);
ksmserver.setGroup("General");
if (ksmserver.readEntry( "loginMode" ) == "restoreSavedSession")
{
    if (KickerSettings::menuEntryHeight() > -1)
    {
        insertItem(SmallIconSet("filesave",
                            KickerSettings::menuEntryHeight()),
                i18n("Save Session"),
                this,
                SLOT(slotSaveSession()));
    }
    else
    {
        insertItem(i18n("Save Session"), this, SLOT(slotSaveSession()));
    }
}
//插入锁屏菜单
if (kapp->authorize("lock_screen"))
{
    if (KickerSettings::menuEntryHeight() > 1)
    {
        insertItem(SmallIconSet("lock", KickerSettings::menuEntryHeight()),
                i18n("Lock Session"),
                this,
                SLOT(slotLock()));
    }
    else
    {
        insertItem(i18n("Lock Session"), this, SLOT(slotLock()));
    }
}
//插入 logout 菜单
if (kapp->authorize("logout"))
{
```

```
        if (KickerSettings::menuEntryHeight() > -1)
        {
            insertItem(SmallIconSet("exit", KickerSettings::menuEntryHeight()),
                    i18n("Log Out..."),
                    this,
                    SLOT(slotLogout()));
        }
        else
        {
            insertItem(i18n("Log Out..."), this, SLOT(slotLogout()));
        }
    }
    setInitialized(true); //设置初始化状态为已初始化
}
```

函数 PanelServiceMenu::initialize()初始化服务菜单，服务菜单的初始化分为 initialize() 和 doInitialize() 两个函数，因为子类不必重做所有的服务分析（如 kicker/menuext/prefmenu）。第 1 个函数列出如下：

```
void PanelServiceMenu::initialize()
{
    if (initialized()) return; //   如果是已初始化状态，返回

    setInitialized(true);  //设置为已初始化状态
    entryMap_.clear();  //清空条目映射链表
    clear();
    subMenus.clear();  //清空子菜单映射链表
    doInitialize();  //态增加菜单条目
}
```

函数 doInitialize 通过应用程序的 KServiceGroup 链表，过滤出符合条件的条目，并动态增加这些条目到菜单中。

在/usr/share/desktop-menu-files 下有服务菜单的桌面配置文件。

函数 doInitialize（在 kdebase-3.4.0/kicker/ui/service_mnu.cpp 中）列出如下：

```
void PanelServiceMenu::doInitialize()
{
    /* 在 panel 外面设置起点位置，这样当我们使用拖动并点击选择条目时，没有被初始化的拖动。一个拖动仅在点击打开菜单并按下拖动
一个条目时被初始化*/
    startPos_ = QPoint(-1,-1);

    //请求 KSycoca 给出所有的服务（排序），放在 root 链表中
    KServiceGroup::Ptr root = KServiceGroup::group(relPath_);

    if (!root || !root->isValid())  //如果服务不存在，返回
        return;
    //从 root 中过滤出满足条件的条目形成链表 list
    KServiceGroup::List list = root->entries(true,
                            excludeNoDisplay_,
                            true,
                            KickerSettings::detailedMenuEntries() &&
                            !KickerSettings::detailedEntriesNamesFirst());

    if (list.isEmpty()) {  //链表为空
        setItemEnabled(insertItem(i18n("No Entries")), false);
        return;
    }

    int id = idStart;

    if (addmenumode_) {
        int mid = insertItem(SmallIconSet("ok"), i18n("Add This Menu"), id++);
        entryMap_.insert(mid, static_cast<KSycocaEntry*>(root));
```

```
    if (list.count() > 0) {
        insertSeparator();
        id++;
    }
}

QStringList suppressGenericNames = root->suppressGenericNames();

KServiceGroup::List::ConstIterator it = list.begin();
for (; it != list.end(); ++it) {

    KSycocaEntry * e = *it;

    if (e->isType(KST_KServiceGroup)) { //如果是服务组，加入子菜单

        KServiceGroup::Ptr g(static_cast<KServiceGroup *>(e));
        QString groupCaption = g->caption();

        // 避免加入空服务组
        KServiceGroup::Ptr subMenuRoot = KServiceGroup::group(g->relPath());
        if (subMenuRoot->childCount() == 0)
            continue;

        // 忽略 '.' (圆点) 文件
        if ((g->name().at(0) == '.'))
            continue;

        //条目名字中含有 '&'，为了避免它们被转换为快捷菜单，将其变成 '&&'
        groupCaption.replace("&", "&&");
        //创建子菜单
        PanelServiceMenu * m =
            newSubMenu(g->name(), g->relPath(), this, g->name().utf8());
        m->setCaption( groupCaption );

        QIconSet iconset;

        if (KickerSettings::menuEntryHeight() > 0)
        {
            iconset = KGlobal::instance()->iconLoader()->loadIconSet(
                      g->icon(), Icon::NoGroup,erSettings::menuEntryHeight());
        }
        else if (KickerSettings::menuEntryHeight() == 0)
        {
            QPixmap normal = KGlobal::instance()->iconLoader()->loadIcon(
                         g->icon(),  KIcon::Small, 0,
                          KIcon::DefaultState, 0L, true);

            QPixmap active = KGlobal::instance()->iconLoader()->loadIcon(
                          g->icon(),  KIcon::Small, 0,
                          KIcon::ActiveState, 0L, true);
            // 确保图标不大于20x20
            if (normal.width() > 20 || normal.height() > 20)
            {
                normal.convertFromImage(normal.convertToImage()
                                        .smoothScale(20,20));
            }
            if (active.width() > 20 || active.height() > 20)
            {
                active.convertFromImage(active.convertToImage()
                                        .smoothScale(20,20));
            }

            iconset.setPixmap(normal, QIconSet::Small, QIconSet::Normal);
            iconset.setPixmap(active, QIconSet::Small, QIconSet::Active);
```

```
            }
            //插入菜单条目
            int newId = insertItem(iconset, groupCaption, m, id++);
            entryMap_.insert(newId, static_cast<KSycocaEntry*>(g));
            // 我们必须自己删除子菜单！（见 Qt 文档。）
            subMenus.append(m);
        }
        else if (e->isType(KST_KService)) {  //插入服务对应的菜单条目
            KService::Ptr s(static_cast<KService *>(e))    ;
            insertMenuItem(s, id++, -1, &suppressGenericNames);
        }
        else if (e->isType(KST_KServiceSeparator)) {
            insertSeparator();  //  插入菜单分隔线
        }
    }
}
```

第 4 章　Qt/Embedded 客户/服务器

在 KDE 桌面系统中，KDE 通过 X Window 的客户/服务器系统完成显示和设备的输入工作。对于嵌入设备来说，由于系统资源的有效（如内存小及 CPU 速度慢），因此嵌入设备的桌面系统不可能建立在 X Window 之上。在这种情况下，Qt/Embedded 给出了自己的客户/服务器模型，并提供了类似 X Window 的功能，完成显示和设备的输入工作。当然，这个模型提供的功能比 X Window 少很多。Qt/Embedded 的客户/服务器模型如图 4-1 所示。

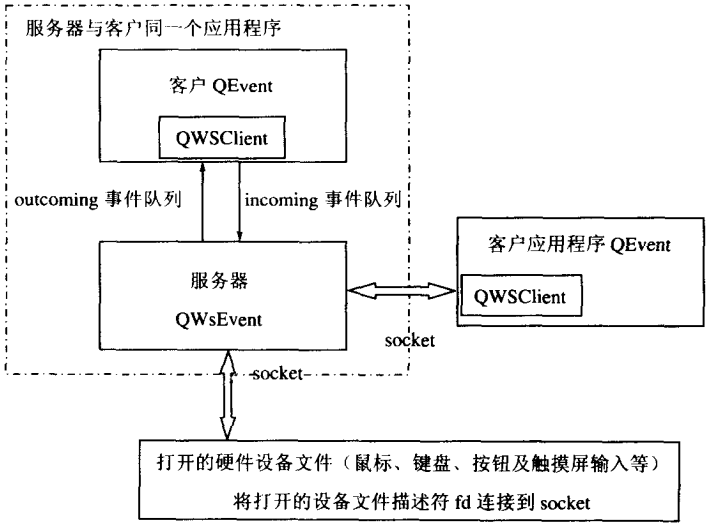


图 4-1　Qt/Embedded 的客户/服务器模型

Qt/Embedded 的客户/服务器模型被封装 Qt/Embedded 核心中，通过 QApplication 建立模型的各种关系，用 Qt/Embedded 类库完成与模型的各种通信及操作。对于应用程序开发者来说，几乎感觉不到 Qt/Embedded 客户/服务器模型的存在，而只需要如同开发 PC 机的 Qt 应用程序一样来开发 Qt/Embedded 应用程序即可。

在 Qt/Embedded 的客户/服务器模型中，第 1 个启动的应用程序会启动服务器。服务器与客户端在同一个应用程序中，后面启动的应用程序仅包括客户端。

本章说明将应用程序下载到嵌入设备运行的方法，介绍各种嵌入式 GUI 窗口系统，阐述 Qt/Embedded 的客户/服务器模型工作过程，分析设备输入事件在客户与服务器之间的传递机制、应用程序窗口的显示原理，并且说明运行 Qt/Embedded 程序的方法。

4.1　各种嵌入式 GUI

嵌入式 GUI 提供了一种交互接口，它要求占用资源小且反应快速，还需要具备高度可移植性与可裁剪性。Linux 系统 GUI 提供了其他图形应用程序使用的基本函数库，这些函数库有 X Window、

SVGALib 和 FrameBuffer 等。在 Linux 中使用的 GUI 有 X Window、MICROWindows、MiniGUI 及 QT/Embedded 等。

4.1.1 X Window

X Window 是类 UNIX 系统中最主要的桌面图形系统，其上运行大量应用程序。 X Window 提供了 DGA （直接图形访问）扩展，应用程序能在全屏模式下直接访问显卡的帧缓冲区，从而满足游戏及多媒体等应用对图形加速能力的要求。 Tiny-X 是 XServer 在嵌入式系统的应用，其目标是运行于小内存系统环境。

在 X Window 中，Xlib 对底层 X 协议进行封装，该函数库实现一般的图形输出。 Xlib 的接口较复杂，高级图形库（如 GTK 及 QT 等）以其为基础。但 GTK 和 QT 等函数库建立在 X Window 上，占用系统资源较多，不适合在嵌入式系统中使用。

4.1.2 MICROWindows

MICROWindows 是一个开放源码的项目，作为一个典型且基于 Server/Clinent 体系结构的 GUI 系统，它基本分为 3 层，如图 4-2 所示。最底层是面向图形输出和键盘、鼠标或触摸屏的驱动程序；中间层提供底层硬件的抽象接口，并进行窗口管理；最高层分别提供兼容 X Window 和 Windows CE （Win32子集）的 API，无任何硬件加速能力。

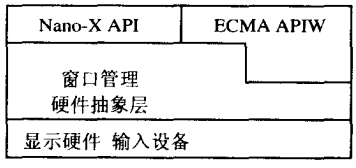


图 4-2 Microwindows 的体系结构

但是该接口没有提供窗口管理，如窗口移动和窗口剪切等高级功能，系统中需要首先启动 nano-X 的 Server 程序 nanoxserver 和窗口管理程序 nanowm。用户程序连接 nano-X 的 Server 获得自身的窗口绘制操作。使用 ECMA APIW 编写的应用程序无需 nanox-server 和 nanowm，可直接运行。

4.1.3 MiniGUI

MiniGUI遵循 LGPL 条款，是基于 Linux 实时嵌入式系统的轻量级图形用户界面系统，其体系框架如图 4-3 所示。MiniGUI 分为最底层的 GAL 层和 IAL 层，向上为基于标准 POSIX 接口中 pthread 库的 Mini-thread 架构和基于 Server/Client 的 Mini-Lite 架构。Mini-thread 架构用于系统功能较为单一的场合； Mini-Lite 应用于多进程的应用场合，实现了各个进程之间的窗口管理及 Z序剪切等。

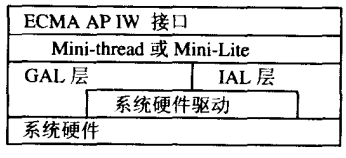


图 4-3 MiniGUI 体系框架

MiniGUI 将硬件设备抽象为 GAL 层和 IAL 层，移植时只需要针对自身的硬件特点按照 GAL 层调

用接口和 IAL 层调用接口来作为内部实现即可。图 4-4 所示为 MiniGUI 的 GAL 层结构示意，IAL 层结构类似。

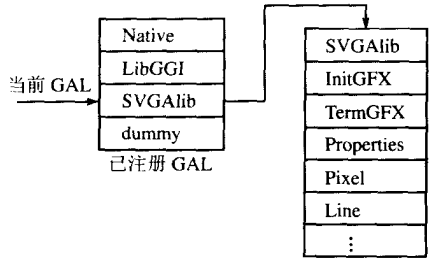


图 4-4 MiniGUI 的 GAL 层结构示意

MiniGUI 的 IAL 层将输入设备的输入事件最终映射为 GUI 系统 API 层的消息事件，IAL 层默认处理两种设备的输入操作，即键盘设备和鼠标设备。键盘设备为上层提供不同的按键输入信息，鼠标设备提供点击、抬起和落笔坐标等的信息。

图形抽象层包含 3 个图形引擎，SVGALib、LibGGI，以及直接基于 Linux FrameBuffer 的 Native Engine。利用 LibGGI 时，可在 X Window 上运行 MiniGUI 应用程序，并可非常方便地进行调试。GAL 能够支持硬件加速能力，并能够充分使用显示内存。GAL 之上的 GDI 接口支持 Alpha 混和、透明位块传输、光栅操作、YUV 覆盖、Gamma 校正，以及高级图形功能（椭圆、多边形及样条曲线）等。

4.1.4 QT/Embedded

Qt/Embedded 是 Trolltech 公司开发的面向嵌入式系统的 Qt 版本，Qt 是 KDE 等项目使用的 GUI 支持库，许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/Embedded 上。Qt/Embedded 是 Server/Client 结构，在底层摒弃了 X lib 仅采用 framebuffer 作为底层图形接口，它是为高端嵌入式图形领域的应用而设计的。同时，将外部输入设备抽象为 keyboard 和 mouse 输入事件，底层接口支持键盘、GPM 鼠标、触摸屏，以及用户自定义的设备等。

Qt/Embedded 类库完全采用 C++封装，兼容同版本的 Qt-X11，使用 X 中的开发工具可以开发基于 Qt/Embedded 的应用程序 QUI 界面。

Qt/Embedded 在体系上是客户/服务器模型，任何一个 Qt/Embedded 程序都可作为系统中惟一的一个 GUI Server 存在。当应用程序首次以系统 GUI Server 的方式加载时，将建立 QWSServer 实例。此时调用 QWSServer::openDisplay() 函数打开显示设备，在 QWSServer::openDisplay() 中调用了 QWSDisplay::Data 中的 init() 函数。在 QWSServer 中，所有关于显示的调用都由 QScreen 派生类完成。当系统中建立 GUI Server 后，其他 Qt/Embedded 程序采用共享内存的进程通信方式，以同步访问模式获得对共享资源 framebuffer 设备的访问权。

Qt/Embedded 的底层显示设备是帧缓冲 /dev/fb0，帧缓冲是标准显示设备驱动接口。使用 mmap 系统调用将帧缓存映射到应用程序虚拟内存空间，这样应用程序可以访问它。

帧缓冲驱动程序的实现分为两个方面，一方面是对 LCD 等相关硬件及缓存的初始化，包括图形在缓冲区的创建和设置 DMA 通道；另外一方面是对画面缓冲区的读写，read、write 及 lseek 等系统调用接口可以调用驱动程序的读写函数。至于将画面缓冲区的内容输出到 LCD 显示屏上，则由硬件自动完成。当设置 DMA 通道和画面缓冲区后，DMA 开始正常工作并将缓冲区中的内容不断发送到 LCD 上，这个过程基于 DMA 对于 LCD 的不断刷新。帧缓冲驱动程序则将数据写入帧缓存中，这通过映射 mmap 来实现。

在 Qt/Embedded 中，Qscreen 类是抽象出的底层显示设备基类。它声明了对于显示设备的基本描

述和操作方式，如打开、关闭、获得显示能力及创建 QGfx 操作对象等。

另外还有一个基类 QGfx 抽象出了显示设备的具体操作接口（图形设备环境），如选择画刷、画线、画矩形及 alpha 操作等。

Qscreen 基类和 QGfx 基类是 Qt/Embedded 图形引擎的底层抽象，其中许多函数是虚函数。当具体的显示设备（如具体的帧缓冲设备和虚拟帧缓冲设备）从其派生类时，这些派生类会继承并重载基类中的虚函数实现。

图 4-5 所示为 QScreen 类及其派生类，从图中可以看出，对于基本的帧缓冲设备， Qt/Embedded 用 QlinuxFbScreen 来处理。具体显示硬件（如 Mach 卡或 Voodoo 卡）具有加速特性，它们从 QlinuxFbScreen 派生类，并根据硬件特性重载虚函数。

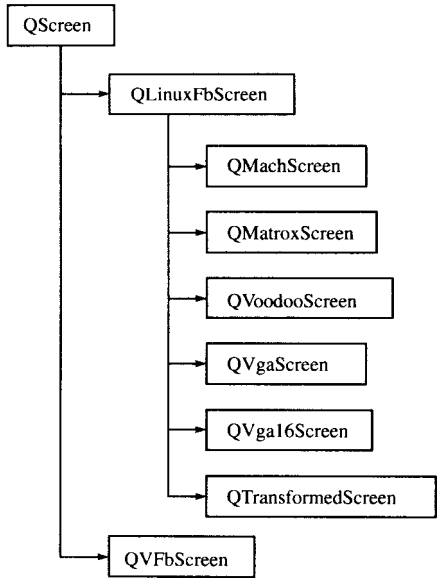


图 4-5 QScreen 类及其派生类

基类 QGfx 抽象出了显示设备的具体操作接口（图形设备环境）， QGfx 类及其的派生类的继承关系如图 4-6 所示。各个具体图形设备的 QScreen 派生类调用函数 createGfx 创建图形设备环境，得到 QGfx 对象，通过 QGfx 对象即可在帧缓冲上绘制基本的图形。 QPixmap 类和 QImage 类通过对 QGfx 类的基本绘图动作进行封装，为用户提供了更高级的图形操作。

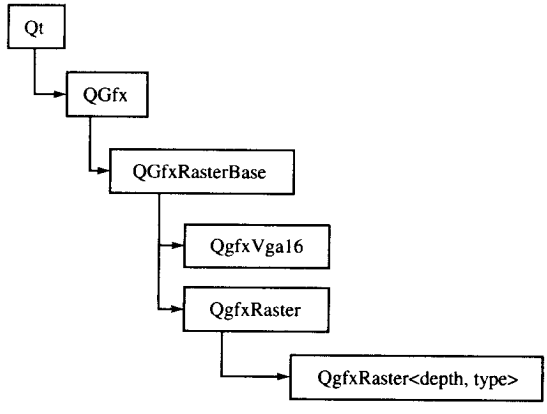


图 4-6 QGfx 类及其派生类的继承关系

函数 createGfx 说明如下：

```
Qfx * QScreen::createGfx(unsigned char * bytes,int w,int h,int d, int linestep)
```

函数 createGfx 在参数 bytes 大小的 buffer（帧缓冲内存的图形区域）中创建一个 QGfx 对象，参数 w 表示宽度（像素点为单位），参数 h 表示高度（像素点为单位），参数 linestep 表示在 buffer 中每一行的长度（byte）。加速的驱动程序能检查 bytes 是否指向图形内存并创建一个加速的 Gfx 对象。

Qt/Embedded 中的输入设备分为鼠标类 QWSMouseHandler 与键盘类 QWSKeyboardHandler，从该类又重新派生出一些特殊鼠标类和键盘类设备的实现类，其派生结构类分别如图 4-7 和图 4-8 所示。键盘类设备接口的基类是 QWSKeyboardHandler。具体的键盘硬件接口依然要建立在键盘驱动程序的基础之上，移植时需要根据键盘驱动程序从该类派生出实现类，实现键盘事件处理函数 processKeyEvent() 即可。

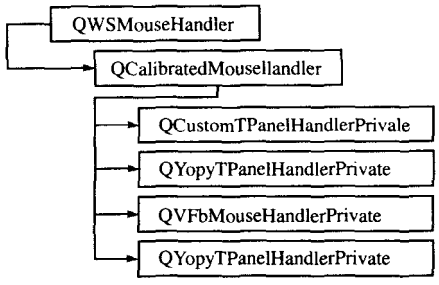


图 4-7 QWSMouseHandler 类派生结构

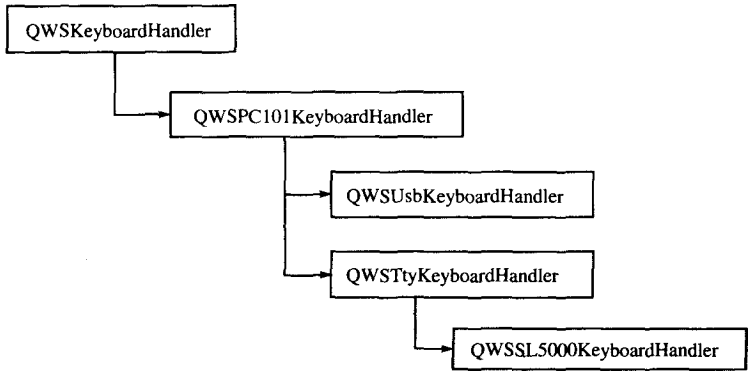


图 4-8 QWSKeyboardHandler 类派生结构

在系统加载构造 QWSServer 时，调用 QWSServer::openMouse 与 QWSServer::openKeyboard 函数。这两个函数分别调用 QMouseDriverFactory::create() 与 QKbdDriverFactory::create 函数。这时会根据 Linux 系统的环境变量 QWS_MOUSE_PROTO 与 QWS_KEYBOARD 获得鼠标类设备和键盘类设备类型和设备节点，打开相应设备并返回相应设备的基类句柄指针给系统。系统通过将基类指令强制转换为对应的具体子类设备指针，获得对具体鼠标类设备和键盘类设备的调用操作。

键盘类设备接口的基类是 QWSKeyboardHandler，具体的键盘设备接口类从该类派生出来，实现了键盘事件处理函数 processKeyEvent()，并在 QWSServer::newKeyboardHandler 函数中注册自己的键盘类设备。对于点击键的键码定义在 Qt/Embedded 的命名空间 src/kernel/qnamespace.h 中。

在键盘点击事件从 QWSServer 截获并发送到相应的 Client 之前会经过函数 QWSServer::KeyboardFilter。在此函数中可以按照自身需求生成新的键盘点击事件，而后利用 QWSServer::sendKeyEvent()发送新的点击事件到 Client 中。利用该方式可以将各种键盘点击无法输入的 Unicode 字

符转换出来，从而可以在较少的按键键盘上实现多 Unicode 字符输入法。

4.2 Qt/Embedded 的应用示例

本例描述了如何安装 Linux 到一个嵌入设备，以及编译一个 Qt/Embedded 应用程序步骤。目标设备是 Cassiopeis E-100/E-105，这个设备使用 Vr4121 处理器，16 MB RAM（在 E-105 上是 32 MB），一个 CF 卡插槽和 240×320 点 16 位的 LCD 显示屏。

嵌入设备的软件包括内核和应用程序部分，本节说明应用程序的生成及安装，而内核的编译过程参见相关书籍。内核和应用程序在台式机 Linux 操作系统下进行交叉编译，内核打包生成 zImage，应用程序打包生成 rootfs_ext2.img。

本例使用内存盘（ramdisk），通过 RAMDISK 控制系统启动运行应用程序。

4.2.1 编译 Linux 内核

为了能够使用 Ramdisk，在编译内核前必须选择 block device 中的 Ramdisk。

4.2.2 root 文件系统

一个 root 文件系统已包括 Linux 的全部系统，即基本文件系统结构至少含有目录/dev、/proc、/bin、/etc、/lib、/usr 和/tmp；基本的应用程序，如 sh、ls、cp 和 mv 等；最低限度的配置文件，如 rc、inittab 和 fstab 等；需要使用的设备文件，如/dev/hd*、/dev/ tty*等，还有基本程序运行所需的函数库。

在台式机用户的默认目录下，建立工作目录 test，shell 命令如下：

```
#mkdir test
```

在 test 目录下用 shell 命令 mkdir 建立以下目录：/dev（用于设备文件）、/proc 和/etc（用于系统配置文件）、/sbin（用于 shell 命令）、/bin （用于基本命令程序）、/lib （用于共享函数库）、/mnt（用于挂接文件系统的节点）及/usr （用于用户的应用程序）。

其中/proc、/mnt 和/usr 是空的，其他目录中需要至少拷贝一些系统基本的文件，分别说明如下。

（1）/dev：拷贝 console、kmem、mem、null、ram 及 tty*等文件。

（2）/etc：拷贝 rc.d/*（系统启动脚本）、fstab （列出要挂接的文件系统）、inittab（包含启动过程参数）、passwd （用户名和目录）、group（用户组）及 shadow （用户加密密码）。

（3）/bin 和/sbin：拷贝 init、getty、login、mount、bash 、ls、mv、cat、cp 及 getty（mingettty）等需要的应用程序，注意它们必须是对应嵌入设备平台运行的二进制程序。

（4）/lib：拷贝函数库装载器 ld.so（对 a.out 库）或 ld-linux.so（对 ELF 库）。

（5）/lib/modules ：拷贝模块 ramdisk.o。

4.2.3 裁剪 Qt/Embedded 库

因为应用程序很简单，所以我们从 Qt/Embedded 库移去不必要的部分。更改$QTDIR/src/tools/qconfig.h 并取消不必须要的部分，宏定义部分更改如下：

```
#define QT_NO_IMAGEIO_BMP
#define QT_NO_IMAGEIO_PPM
#define QT_NO_ASYNC_IO
#defne QT_NO_BDF
```

```
#define QT_NO_FONTDATABASE
#define QT_NO_WIZARD
```

查看 Qt 特征描述，取消不必须的 widget。在 x86 平台上为 mips 目标设备交叉编译 QT 库，方法如下：

```
cd $QTDIR
./configure -xplatform linux-mips-g++ -platform linux-x86-g++
make mipsel-linux-strip $QTDIR/lib/libqt.so.2.2.0
```

4.2.4 编译应用程序

编译应用程序的命令如下：

```
cd examples/dclock
make mipsel-linux-strip dclock
```

我们已选择了动态连接应用程序，如果需要运行多个应用程序，这是重要的。如果仅有一个应用程序运行，也可以选择静态连接。静态连接配置如下：

```
./configure -static -xplatform linux-mips-g++ -platform linux-x86-g++
```

4.2.5 拷贝 Qt 库及应用程序

假定 loop0 设备挂接的目录为/mnt/loop，拷贝 Qt/Embedded 库到/mnt/loop/lib 目录下，并创建需要的链接：

```
cd /mnt/loop/lib
cp $QTDIR/lib/libqt.so.2.2.0
ln -s libqt.so.2.2.0 libqt.so.2.2
ln -s libqt.so.2.2.0 libqt.so.2
```

字体必须安装在/usr/local/qt-embedded/etc/fonts 下：

```
cd /mnt/loop
mkdir usr/local
mkdir usr/local/qt-embedded/etc
mkdir usr/local/qt-embedded/etc/fonts
cp helvetica_100_50.qlf Helvetica_120_50.qlf /usr/local/qt-embedded/etc/fonts
```

当内核启动时，它查找多个文件来运行。为了运行应用程序，需要把它重命名到 /bin/sh，Qt/Embedded 还用到/tmp 目录：

```
cp $QTDIR/examples/notepad/dclock /mnt/loop/bin/sh
mkdir /mnt/loop/tmp
umount /mnt/loop
```

4.2.6 打包

打包的过程如下。

（1）建立临时文件/tmp/loop_tmp

/dev/zero 文件代表一个永远输出 0 的设备文件，使用它作为输入可以得到全为 0 的文件，因此可用来创建新文件并以覆盖方式清除旧文件。

设备文件/dev/null 表示永远无法写满内容的设备，写入的内容被系统立即丢弃。如果不想看到程

序的输出，可以使用它作为输出。它常用来去除一些操作对屏幕输出信息。

下面使用 dd 命令将从 zero 设备中创建一个 10 MB 内容全为 0 临时文件，其中 bs 表示每次读写 1 024 个字节，count 表示读写次数为 1 000 次，即为 10 MB，命令列出如下：

```
#dd if=/dev/zero of=/tmp/loop_tmp bs=1k count=1000 >/dev/null
```

（2）建立 loop 设备的临时挂接点

使用如下命令建立 loop 设备的临时挂接点：

```
#mkdir /mnt/loop 2>/dev/null
```

（3）将 loop 设备与临时文件联系起来

由于要用到 loop 设备，台式机 Linux 内核应支持这个设备。loop 设备将文件作为一个文件系统，可以执行 mount/umount 操作。这样即可经由 loop 设备，把代表 ext2 文件系统的 rootfs_ext2.img 文件 mount 到/tmp/loop_tmp 目录上。

使用下面的命令将 loop 设备与临时文件联系起来：

```
#losetup /dev/loop0 /tmp/loop_tmp
```

（4）loop 设备格式化

Linux 内核识别两种可以直接拷贝到 RAMDISK 的文件系统，即 minix 和 ext2。其中 ext2 性能更好，因而这里选择使用 ext2 文件系统。使用 mke2fs 对 loop0 设备进行格式化文件系统，命令参数-m 0 防止它为 root 保留空间，这样会腾出更多的有用空间。操作命令列出如下：

```
#mke2fs -m 0 /dev/loop0 2>/dev/null
```

（5）把 loop0 设备挂接在/mnt/loop 上

使用如下命令把 loop0 设备挂接在/mnt/loop 上：

```
#mount /dev/loop0 /mnt/loop -t ext2
```

（6）拷贝 root 文件系统

将制作的 root 文件系统拷贝到/mnt/loop 中，卸下挂接点，删除建立的挂接点。然后取消 loop 设备与临时挂接点的联系，操作命令依次列出如下：

```
#cp -a ./tree/* /mnt/loop
#umount /mnt/loop
#rmdir /mnt/loop 2>/dev/null
#losetup -d /dev/loop0
```

（7）压缩文件系统得到 rootfs.gz

压缩文件系统后，就拥有了一个压缩的 root 文件系统 rootfs.gz，删除建立的临时文件。压缩文件系统的命令如下：

```
#dd if=/tmp/loop_tmp | gzip -9 > rootfs.gz
#rm -f /tmp/loop_tmp
```

4.2.7 下载及运行

创建一个 ramdisk 对象，把它与内核连接在一起。然后拷贝到闪存中并启动 Linux，应该可以看到屏幕上出现 dclock 应用程序的界面。

4.3 帧缓冲

嵌入设备的显卡驱动都使用帧缓冲驱动程序，在台式机上为了运行嵌入设备的应用程序，可以使用虚拟帧缓冲 qvfb 来模拟嵌入设备的图形环境。

嵌入设备还可以通过 DirectFB 来在帧缓冲驱动程序中进行硬件加速。

4.3.1 DirectFB 介绍

DirectFB 是一个小型库，它提供了在 Linux 帧缓冲设备上的硬件图形加速、输入设备处理和支持抽象，并且支持透明和多个显示层的集成窗口系统。它是一个完全的硬件抽象层，对于硬件加速来说，DirectFB 使用可加载的驱动程序模块进行硬件加速，并使用软件实现不被底层硬件支持的每个图形操作。DirectFB 为嵌入系统设计，它提供了最大的硬件加速效率最小的资源使用。

DirectFB 依赖于存在的内核接口访问图形硬件，帧缓冲设备提供了设备文件/dev/fb，在某些芯片中需要有特殊的帧缓冲驱动。但 VESA 帧缓冲驱动还是可以工作，尽管有一些限制。

为了运行特定的图形操作，DirectFB 芯片驱动程序将访问图形硬件的内存映射的 IO 端口，提供卡的加速引擎的命令，这意味着实际的硬件加速都是从用户空间进行的。当 DirectFB 应用程序运行时，它通过帧缓冲驱动程序控制显卡寄存器。数据直接传入帧缓冲映射中，并通过用户空间的芯片驱动模块来控制硬件加速。DirectFB 应用程序访问帧缓冲设备和图形硬件的流程如图 4-9 所示。

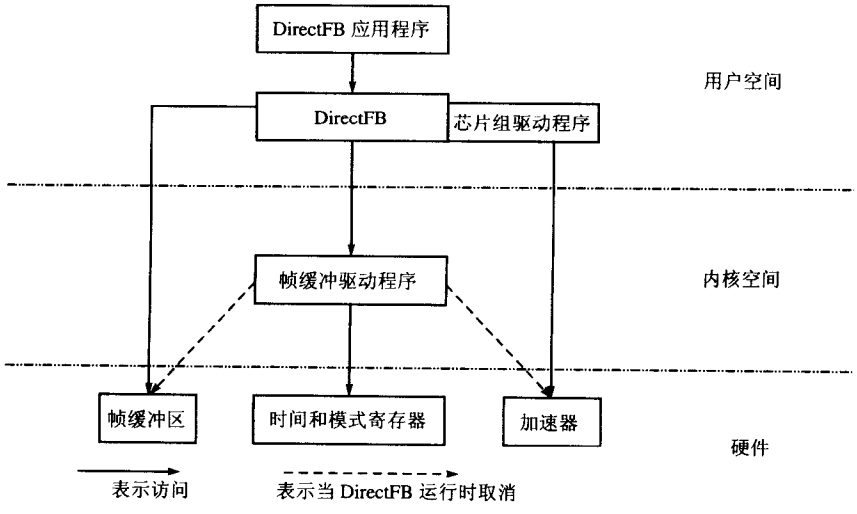


图 4-9 DirectFB 应用程序访问帧缓冲设备和图形硬件的流程

对于访问输入设备，DirectFB 使用了 Linux 内核提供的标准设备接口，它不能直接访问输入硬件。

4.3.2 虚拟帧缓冲 qvfb

虚拟帧缓冲允许 Qt/Embedded 程序在台式机 Linux环境下运行，即将 Qt/Embedded 应用程序编译为 x86平台版本，使其可以在台式机 Linux 下的 qvfb 窗口中运行。在应用程序调试好后，可以编译为 xscale 平台版本，下载到嵌入设备中运行。这样，方便了应用程序的编写与调试。虚拟帧缓冲 qvfb 在 $QTDIR/tools/qvfb 目录下。

使用虚拟帧缓冲 qvfb 的方法如下：

（1）确信在$QTDIR/src/tools/qdonfig.h 中 QT_NO_QWS_VFB 尚未定义，并且编译 t/EmbeddedQ 库。

（2）将 qvfb 作为正常 x86 平台的 Qt/X11 应用程序编译，而不要作为 Qt/Embedded 应用程序编译。生成二进制应用程序后，以如下命令运行它：

```
qvfb
```

当 qvfb 应用程序窗口出现时，可从窗口菜单中选择设置 qvfb 的显示颜色深度及显示尺寸。对于 PDA 来说，一般设置为 16 位颜色深度、320×240 像素点显示区域，即与通常的 PDA 屏幕完成一样。

（3）启动一个编译后的二进制 Qt/Embedded 应用程序，命令行中使用 -qws 表示将应用程序作为 Qt/Embedded 服务器，还可以使用参数选项-display QVFb:0 指定显示服务器。也可以通过将环境变量 QWS_DISPLAY 设置为 QVFb:0 来指定显示服务器，启动应用程序的命令如下：

```
test -qws -display QVFb:0  #假定应用程序为 test
```

这时，在 qvfb 的窗口中应该能看到应用程序的运行窗口。

虚拟帧缓冲 qvfb 使用共享内存区域模仿一个 framebuffer （帧缓冲），并在一个窗口中显示 framebuffer。显示的区域周期性地被更新，这样可看到具体的帧缓冲的快照，而不是每个单独的绘制操作。因为这个原因，绘制中的问题（如闪烁）将不会出现，直到使用真实的帧缓冲来运行程序时才发现。

目标刷新速率通过 qvfb 工具的 Viewl Refresh Rate 菜单项来设置，它将引起 qvfb 更快地检查更新区域，这个速率只是一个目标而已。如果几乎没有绘制操作，帧缓冲将不显示两次绘制事件之间任何更新。如果应用程序显示一个动画，更新将更为频繁，并且应用程序和 qvfb 将竞争处理器时间。

鼠标和键盘事件将通过命名管道被传递到 Qt/Embedded 主进程，qvfb 仅是一个开发工具。它应避免出现在产品环境中，QT_NO_QWS_VFB 应该总定义在作为产品的库中。

4.4 创建应用程序

Qt 应用程序运行时，在 main 函数中调用 QApplication 类构造函数来管理应用程序，本节分析 QApplication 类及构造函数。

4.4.1 QApplication 类

QApplication 类管理 GUI 应用程序的控制流和主要设置，其中包含主事件循环，来自窗口系统和其他资源的所有事件都在主事件循环中被处理和分发。它还处理应用程序的初始化和结束，提供会话管理，并且处理大多数系统范围和应用程序范围的设置。

使用 Qt 的 GUI 应用程序只能存在一个 QApplication 对象，即使这个应用程序在同一时间内有多个窗口。

通过全局变量 qApp 访问 QApplication 对象，它负责的主要范围如下。

（1）使用用户的桌面设置，例如 palette()、font()和 doubleClickInterval()，可以初始化应用程序。并且保持跟踪用户改变整体桌面的属性，例如通过一些控制面板改变桌面的整体属性。

（2）执行事件处理，从底层窗口系统接收事件并且把它们分发给相关的窗口部件。通过使用 sendEvent() 和 postEvent()，可以发送事件到窗口部件。

（3）分析命令行参数并设置 qApp 相应的内部状态。

（4）定义由 QStyle 对象封装的应用程序的外观风格，在运行状态下，可以通过 setStyle()改变。

（5）指定应用程序如何分配颜色。

（6）提供通过 translate()用户可见的本地化字符串。

（7）提供一些诸如 desktop()和 clipboard()这样的对象。

（8）知道应用程序的窗口，即可以使用 widgetAt()询问在一个确定点上存在哪个窗口部件，得到一个topLevelWidgets()（顶级窗口部件）和 closeAllWindows()的列表等。

（9）管理应用程序的鼠标光标处理，如 setOverrideCursor()和 setGlobalMouseTracking()。

（10）在 X 窗口系统中提供刷新和同步通信流的函数 flushX()和 syncX()。

（11）提供复杂的会话管理支持，这使得当用户注销时关闭应用程序成为可能，撤销关闭进程。如果不能结束进程，则可以取消关闭进程，甚至为将来的会话保存整个应用程序的状态。与会话相关的函数是 isSessionRestored()、sessionId()、commitData()和 saveState()。

因为 QApplication 对象做了如此多的初始化，所以它必须在创建所有与用户界面相关的其他对象之前创建。

因为处理命令行参数，所以在应用程序中对 argv 解释和修改之前创建它通常是一个好主意。注意，对于 X11，setMainWidget()可以根据-geometry 选项来改变主窗口部件。为了保持这个功能，必须在 setMainWidget()及其任何重载之前设置默认。

QApplication 类的成员函数说明如表 4-1 所示。

表 4-1 QApplication 类的成员函数说明

函数分类	函数分组
系统设置	desktopSettingsAware()、setDesktopSettingsAware()、cursorFlashTime()、setCursorFlashTime()、doubleClickInterval()、setDoubleClickInterval()、wheelScrollLines()、setWheelScrollLines()、palette()、setPalette()、font()、setFont()和 fontMetrics()
事件处理	exec()、processEvents()、enter_loop()、exit_loop()、exit()、quit()。 sendEvent()、postEvent()、sendPostedEvents()、removePostedEvents()、hasPendingEvents()、notify()、macEventFilter()、qwsEventFilter()、x11EventFilter()、x11ProcessEvent()和 winEventFilter()
图形用户界面风格	style()、setStyle()和 polish()
颜色使用	colorSpec()、setColorSpec()和 qwsSetCustomColors()
文本处理	setDefaultCodec()、installTranslator()、removeTranslator()和 translate()
窗口部件	mainWidget()、setMainWidget()、allWidgets()、topLevelWidgets()、desktop()、activePopupWidget()、activeModalWidget()、clipboard()、focusWidget()、winFocus()、activeWindow()和 widgetAt()
高级光标处理	hasGlobalMouseTracking()、setGlobalMouseTracking()、overrideCursor()、setOverrideCursor()和 restoreOverrideCursor()
X 窗口系统同步	flushX()和 syncX()
对话管理	isSessionRestored()、sessionId()、commitData()和 saveState()
线程	lock()、unlock()、locked()、tryLock()和 wakeUpGuiThread()
杂项	closeAllWindows()、startingUp()、closingDown()和 type()

4.4.2 QApplication 构造函数

QApplication 类的构造函数启动 qwsServer，建立与帧缓冲服务器的连接 Display，并创建与会话管理器的连接等。对应用程序的通用管理大部分在类 QApplication 的构造函数中完成，下面进行分析。

构造函数 QApplication 构造一个应用程序对象，参数 type 是应用程序类型。对于 Qt/Embedded 来说，如果应用程序作为服务器使用，则 type 应传入 QApplication::GuiServer 。构造函数 Qapplication 的调用层次如图 4-10 所示。

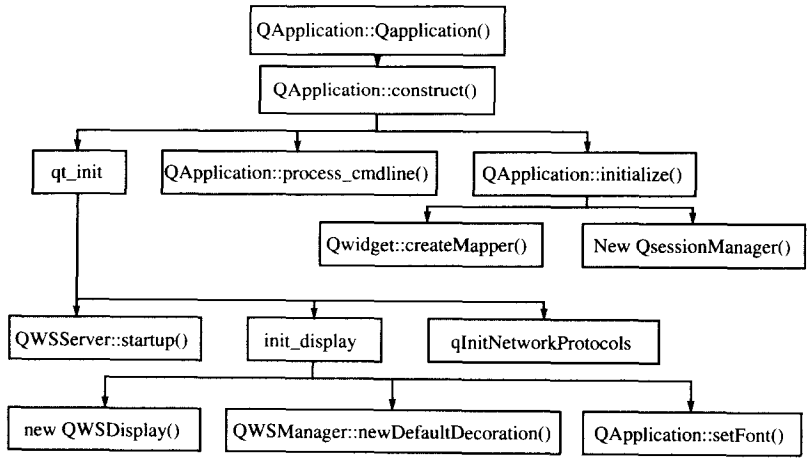


图 4-10 构造函数 Qapplication 的调用层次

构造函数 QApplication 列出如下：

```
QApplication::QApplication( int &argc, char **argv, Type type )
{
    construct( argc, argv, type );
}

void QApplication::construct( int &argc, char **argv, Type type )
{
    ......
    qt_init( &argc, argv, type );  // Must be called before initialize()
    process_cmdline( &argc, argv );
    initialize( argc, argv );
    if ( qt_is_gui_used )
        qt_maxWindowRect = desktop()->rect();  //最大窗口尺寸为桌面窗口尺寸
}
```

函数 qt_init 读取环境变量 QWS_SW_CURSOR 和 QWS_DISPLAY、分析命令行参数、启动 qwsServer 服务器，创建并初始化与帧缓冲（FB）服务器的连接 Display，并初始化了网络协议。

函数 qt_init（在 src/kernel/qapplication_qws.cpp 中）列出如下：

```
void qt_init( int *argcptr, char **argv, QApplication::Type type )
{
    ...
    const char *display = getenv("QWS_DISPLAY");
    ......
        QCString arg = argv[i];
        if ( arg == "-fn" || arg == "-font" ) {
            if ( ++i < argc )
                appFont = argv[i];
        }
    ......

    if ( type == QApplication::GuiServer ) {
            qt_appType = type;
            qws_single_process = TRUE;
            QWSServer::startup(flags); //启动 qwsServer 服务器
            setenv("QWS_DISPLAY", qws_display_spec, 0);
    }

    if( qt_is_gui_used ) //对于服务器来说，已调用了这个函数，则不需要再调用
                init_display(); //创建并初始化与帧缓冲（FB）服务器的连接 Display
```

```
#ifndef QT_NO_NETWORK
    qInitNetworkProtocols(); //初始化网络协议
#endif
}
```

函数 initialize 初始化了 QApplication 对象，它创建调色板，并连接到会话管理器函数 initialize。在 QApplication 构造函数中被调用，该函数列出如下：

```
void QApplication::initialize( int argc, char **argv )
{
#ifdef QT_THREAD_SUPPORT  //创建线程互斥变量
    qt_mutex = new QMutex( TRUE );
    postevent_mutex = new QMutex( TRUE );
    qt_application_thread_id = QThread::currentThread();
#endif // QT_THREAD_SUPPORT

    app_argc = argc;
    app_argv = argv;
    quit_now = FALSE;
    quit_code = 0;
    QWidget::createMapper(); // create widget mapper
#ifndef QT_NO_PALETTE
    (void) palette();  // 创建应用程序调色板
#endif
    is_app_running = TRUE; //应用程序正在运行，不再启动

#ifndef QT_NO_SESSIONMANAGER
    // 连接到会话管理器
    if ( !session_key )
        session_key = new QString;
    session_manager = new QSessionManager( qApp, session_id, *session_key );
#endif

}
```

4.5 构建服务器

在 Qt 应用程序中只有一个用于服务器，一般是第 1 个启动的 Qt 应用程序。与其他应用程序的区别是它创建了服务器 qwsSever，并且只支持单线程（即 qws_single_process 为 TRUE）。qwsSever 是类 QWSServer 的一个全局实例。

4.5.1 QWSServer 类说明

QWSServer 类管理连接到客户端 Unix 域 socketQt/Embedded 的服务器，它发送或接收 QWS 协议事件，并且调用 QWSDisplay 完成诸如改变窗口区域分配的工作。一般不需要修改此类。

QWSServer 提供了在 Qt/Embedded 中服务器特定的功能，当运行 Qt/Embedded 应用程序时，这个应用程序可作为一个服务器运行或连接到一个存在的服务器。如果作为一个服务器运行，一些附加操作通过 QWSServer 类来提供。

对于 Qt/Embedded 服务器进程来说，QWSServer 类被 QApplication 来实例化，你不能自己构造这个类的实例。

通过全局变量 qwsServer 可以得到 QWSServer 实例，其成员函数 openMouse() 和 openKeyboard() 用来打开鼠标和键盘设备，closeMouse()和 closeKeyboard() 关闭鼠标和键盘设备。refresh()用来刷新屏幕，enablePainting() 用来激活或关闭绘制屏幕。clientWindows()用来获得客户窗口的链表，windowAt()

找到一个特定点所在的窗口。另外，还提供了许多静态函数，如 setKeyboardFilter()，setKeyboardHandler()，setDefaultKeyboard()和 setDefaultMouse()。

还有其成员函数 setMaxWindowRect() 可设置窗口的方框大小，setDesktopBackground()设置桌面的背景，setScreenSaverInterval() 设置屏幕保护的间隔时间，screenSaverActivate()激活屏幕保护。

构造函数 QWSServer()构造了一个 QWSServer 类实例，它在 Qt/Embedded 服务器进程的 QApplication 中被实例化，不应该由开发者来构造实例。参数 flag 用于键盘和鼠标的设置，参数 parent 是服务器的父类对象指针，参数 name 是服务器名。

4.5.2 QWSServer 类构造函数分析

构造函数 QWSServer()的功能是设置 server/client 通信方式、打开到帧缓冲服务器的连接、唤醒屏幕保护、创建客户端、打开键盘和鼠标、初始化光标，并创建声音服务器。 QWSServer 类构造函数的调用层次图如图 4-11 所示。

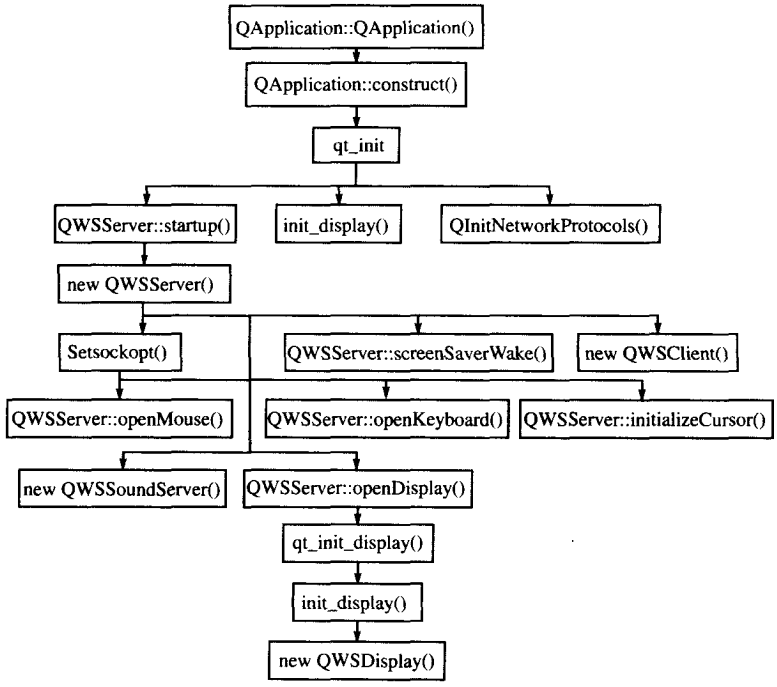


图 4-11 QWSServer 类构造函数的调用层次

该构造函数（在 qtembededed/src/kernel/qwindowsystem_qws.cpp 中）列出如下：

```
//全局变量声明，在其他类中使用这个变量来访问 QWSServer 实例
QWSServer *qwsServer=0;

QWSServer::QWSServer( int flags, QObject *parent, const char *name )
#ifndef QT_NO_QWS_MULTIPROCESS
   //创建本地 socket，绑定到文件 qws_qtePipeFilename()上并侦听 socket
   QWSServerSocket(qws_qtePipeFilename(),16,parent,name),
#else
   QObject( parent, name ),
#endif
   disablePainting(FALSE)
{
   /*存有 QWSServer 的数据，如鼠标状态、屏保时间、 QWSScreenSaver类指针和 QWSClient 类指针等*/
```

```
    d = new QWSServerData;
    Q_ASSERT( !qwsServer ) ;
    qwsServer = this;  //将此实例赋给了全局变量

#ifndef QT_NO_QWS_MULTIPROCESS
    //得到 Qt/Embedded用于 server/client 通信管道的文件名
    QString pipe = qws_qtePipeFilename();

    if ( !ok() )  {
        perror("Error")  ;
        qFatal("Failed to bind to %s", pipe.latin1() )    ;
    } else {
        struct linger tmp;
        tmp.l_onoff=1;
        tmp.l_linger=0;
    /*C库函数,设置 socket 选项。socket()表示一个打开的套接口描述子，选项可能存在多个协议级别上，SOL_SOCKET 表示处理 socket
级别上的选项。选项名为 SO_LINGER,值 为 tmp*/
        setsockopt(socket(),SOL_SOCKET,SO_LINGER,(char *)&tmp,sizeof(tmp));
    }

    signal(SIGPIPE, ignoreSignal); //安装处理函数到信号 SIGPIPE
#endif
    focusw = 0;
    mouseGrabber = 0;
    mouseGrabbing = FALSE;
    keyboardGrabber = 0;
    keyboardGrabbing = FALSE;
#ifndef QT_NO_QWS_CURSOR
    haveviscurs = FALSE;
    cursor = 0;
    nextCursor = 0;
#endif

#ifndef QT_NO_QWS_MULTIPROCESS

    if ( !geteuid() ) {
#if !defined(Q_OS_FREEBSD) && !defined(Q_OS_SOLARIS) && !defined(Q_OS_MACX)
        //挂接共享文件内存系统 (shmfs) 到/var/shm
        if( mount(0,"/var/shm", "shm", 0, 0) ) {

        }
#endif
    }
#endif

    //在还没有进行窗口选择时初始化 selectionOwner 结构
    selectionOwner.windowid = -1;
    selectionOwner.time.set( -1, -1, -1, -1 );

    openDisplay(); // 打开显示，即建立显示相关驱动、framebuffer 等

    d->screensavertimer = new QTimer(this); //建立屏幕保护定时器
    connect( d->screensavertimer, SIGNAL(timeout()), this,
                                                SLOT(screenSaverTimeout()) );
    screenSaverWake(); //唤醒屏幕保护

    /*创建客户端 QWSClient，参数-1 表示是服务器，全局变量 CSocket 为 0  由于已建立了 QWSServerSocket，所以不需要再建立
socket*/
    client[-1] = new QWSClient( this, -1, 0 );

    //输入设备
    if ( !(flags&DisableMouse) )  {
        openMouse();  //  打开鼠标
    }
    initializeCursor();  //设置鼠标箭头形状及在屏幕中间位置
```

```
#ifndef QT_NO_QWS_KEYBOARD
    if ( !(flags&DisableKeyboard) ) {
        openKeyboard();  //打开键盘设备
    }
#endif
    if ( !bgColor )
        bgColor = new QColor( 0x20, 0xb0, 0x50 ); //设置背景颜色值
    screenRegion = QRegion( 0, 0, swidth, sheight );  //设置屏幕区域
    paintBackground( screenRegion ); //画背景颜色

#if !defined( QT_NO_SOUND ) && !defined( Q_OS_MACX )
    soundserver = new QWSSoundServer(this); //创建声音服务器
#endif
#ifndef QT_NO_QWS_IM
    microF = FALSE;
#endif
}
```

函数 openDisplay 打开一个到帧缓冲服务器的连接，得到图形区域管理器和用于在帧缓冲区上绘制基本图形的 QGfx 对象，列出如下：

```
void QWSServer::openDisplay()
{
    qt_init_display();

    rgnMan = qt_fbdpy->regionManager();
    swidth = qt_screen->deviceWidth();
    sheight = qt_screen->deviceHeight();
    gfx = qt_screen->screenGfx();
}
```

函数 qt_init_display（在 qt-embedded/src/kernel/qapplication_qws.cpp 中）初始化到帧缓冲服务器的连接，列出如下：

```
QWSDisplay*    qt_fbdpy = 0;  // 全局变量指针
void qt_init_display()
{
    qt_is_gui_used = TRUE;  //不能再调用这个函数
    qws_single_process = TRUE;  //只支持单线程
    init_display();
}
```

函数 init_display 创建一个到帧缓冲服务器的连接，并初始化颜色、字体、光标及调色板等方面的数据，列出如下：

```
static void init_display()
{
    if ( qt_fbdpy ) return; // workaround server==client case

    setlocale( LC_ALL, "" );        //使用正确的字符集映射
    setlocale( LC_NUMERIC, "C" );  //使用 sprintf()/scanf() 工作

    // 连接到 FB(frame buffer) server
    qt_fbdpy = new QWSDisplay();

    QColor::initialize();           //设置颜色深度
    QFont::initialize();            // 创建全局的字体 cache
#ifndef QT_NO_CURSOR
    QCursor::initialize();          //初始化鼠标箭头的形状及位置（在屏幕中间）
#endif
    QPainter::initialize();         //函数为空
```

```
#if defined(QT_THREAD_SUPPORT)
//内部用，初始化 QThread 系统
    QThread::initialize();
#endif
    //创建一个新的 QFontManager 实例，即 qt_fontmanager=new QFontManager();
    QFontManager::initialize();
#ifndef QT_NO_QWS_MANAGER
    qws_decoration = QWSManager::newDefaultDecoration();
#endif

    incoming.setAutoDelete(true);
    outgoing.setAutoDelete(true);

    qApp->setName( appName );

    QFont f;
    f = QFont( "helvetica", 10 );
    QApplication::setFont( f );  //设置字体

    qt_set_qws_resources();      // 设置标准调色板，及调色板中激活的前景、背景和按钮颜色
}
```

4.6 Qt/Embedded 客户端与服务器通信

Qt/Embedded 的客户/服务器模型与一般的客户/服务器模型概念不完全一致，其特点是客户端和服务器都是 Qt 应用程序。服务器不是独立运行的，而是附加在客户端应用程序上运行的。因此 Qt/Embedded 的代码中，客户和服务器的代码在上层都基本上在一个函数中处理，用一些变量来标识是否是服务器用的代码。

客户端可有多个，但服务器只有一个。因此当第 1 个 Qt 应用程序运行时，它成为服务器。它除了具有一般 Qt 应用程序的客户端外，还增加了服务器的功能，包括初始化鼠标、键盘及显示设备等，并且建立代表显示设备及相关数据结构的 DISPLAY 。另外，服务器还对所有窗口进行全局管理。

客户/服务器间的通信原理如图 4-12 所示。

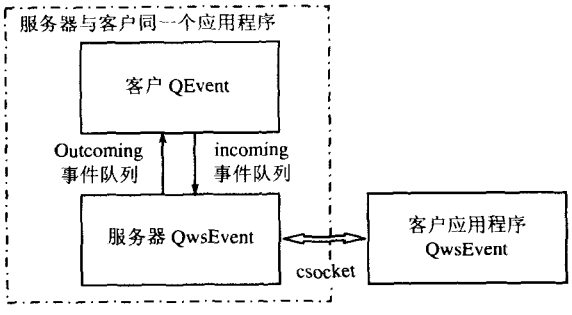


图 4-12　客户/服务器间的通信原理

不同进程间的客户/服务器间通信通过 socket 文件描述符 csocket 进行，客户端及服务器通过函数 QWSClient:sendEvent()向对方发送事件。

客户/服务器间通信说明如下。

（1）服务器端

对于多进程来说，服务器将来自鼠标及键盘等硬件输入的数据通过 socket 以事件形式发送给客户

端，另外还把服务器对窗口的全局管理方面的事件发送给客户端。同时，服务器还通过 socket 接收来自客户端的命令，并激活相关操作函数。对于单个进程来说，因为在同一个进程空间中，所以只需要通过一个队列结构即可传递事件。

（2）客户端

对于多进程来说，客户端将一些需要服务器处理的窗口管理及鼠标键盘命令数据通过 socket 以事件形式发送给服务器。同时，客户端还通过 socket 接收来自服务器的事件，分发给客户端相关的窗口进行处理。对于单个进程来说，因为在同一个进程空间中，所以事件只需要通过一个队列结构即可传递。

4.6.1 服务器相关的类说明

服务器对所有窗口进行全局性管理，如使用 winID 标识所有窗口等，这通过 QWSWindow 类进行。客户端需要建立 socket，为此使用了 QWSClient 类，服务器使用 QWSSeverSocket 类来建立 socket。在客户/服务器间传递的事件通过 QWSEvent 类描述，传递的命令通过 QWSCommand 类来描述。下面对这些类分别进行说明。

（1）QWSServer 类

QWSServer 类代表 Qt/Embedded 服务器，它管理鼠标和键盘设备、初始化显示设备、激活或关闭绘制屏幕、获得客户窗口的链表，并且设置窗口的方框大小等。

（2）QWSWindow 类

QWSWindow 类把 QWSServer 类作为友类，它代表一个窗口，保存 QWSClient 类指针及窗口的名字、标题、区域及可视性等属性的参数。该类可设置或得到窗口的区域及可视属性，还提供了 Qt/Embedded 中服务器特定的功能。

通过 QWSWindow 类的成员函数 name()可得到这个窗口的名字，caption()得到窗口的标题，winId()得到窗口的 winID，client()得到拥有这个窗口的客户端。函数 requested()返回窗口请求绘制的区域，函数 allocation()指定允许绘制是的部分，函数 isVisible()、isPartiallyObscured() 和 isFullyObscured()决定窗口的可视部分。通过 raise()、lower()、show()、hide()和 setActiveWindow()可以改变窗口的可视性。

（3）QWSClient 类

QWSClient 类代表一个客户端，用于不同 Qt 应用程序客户端发送事件或应用程序内部客户发送事件给服务器。它创建 socket，并通过 socket （对于多线程服务器）发送事件，或通过全局的 QWSEvent 对象队列（对于单线程服务器）发送事件。它还发送窗口属性改变相关的事件（如焦点、窗口最大化、选中窗口或区域等），发送事件的一个基本函数是函数 sendEvent(QWSEvent* event)。

QWSClient 类是 Qt 内部使用的类，在其构造函数中创建了一个 QWSSocket 对象，并连接其 readyRead()信号、connectionClosed()信号和 error(int)信号。3 个信号说明如下。

- void QSocket::readyRead ()[信号]：当可以读取进入的数据时，这个信号被发射。每一次有新进入的数据时，这个信号都被发射一次。新进入的数据只被报告一次，即如果没有读取全部数据，这个信号不会被再次发射；除非新的数据到达这个套接字。
- void QSocket::connectionClosed ()[信号]：当另一端已经关闭这个连接时，这个信号被发射。在连接被关闭之后，读缓存中也许还包含可读缓存的输入数据。
- void QSocket::error(int)[信号] 在错误发生之后，信号被发射。参数是 Error 的值。
- QWSEvent 类

服务器事件 QWSEvent 结构描述了服务器事件的种类，如 Mouse、Focus、Key 和 QCopMessage

等。其父类 QWSProtocolItem 保存事件数据的原始流，并提供成员函数 write 将原始流数据写入 socket，提供成员函数 read 从 socket 读出原始流数据。

各个具体服务器事件类从 QWSEvent 派生，这些具体的事件类在父类结构 QWSEvent 的基础上加入了具体数据结构 simpleData，该结构解析了 QWSProtocolItem 保存事件数据的原始流。

例如：服务器鼠标事件的结构（在 src/kernel/qwsevent_qws.h 中）定义如下：

```
struct QWSMouseEvent : QWSEvent {
    QWSMouseEvent()
        : QWSEvent( QWSEvent::Mouse, sizeof( simpleData ),
                (char*)&simpleData ) {} //作为字符原始流数据
    struct SimpleData {
        int window;  //窗口号
        int x_root, y_root, state; //鼠标的座标 x，y 和状态
        int time; //单位为 ms
    } simpleData;
};
```

（5）QWSCommand 类

QWSCommands 包裹了发送/接收 QWS 服务器的数据，不应该修改它。QWSCommand 结构描述了服务器命令的种类，如 Create、RegionMove、SetProperty 和 QCopSend 等，其父类为 QWSProtocolItem。各个具体的服务器命令类从 QWSCommand 派生，并使用了结构 simpleData 来解释原始数据流。

例如，服务器设备属性的命令结构（在 src/kernel/qwscommand_qws.h 中）列出如下：

```
struct QWSSetPropertyCommand : public QWSCommand
{
    QWSSetPropertyCommand() :
        QWSCommand( QWSCommand::SetProperty, sizeof( simpleData ),
                    (char*)&simpleData ) { data = 0; }
    //设置数据
    void setData( char *d, int len, bool allocateMem = TRUE ) {
          QWSCommand::setData( d, len, allocateMem );
          data = rawDataPtr;
    }

    struct SimpleData {
          int windowid, property, mode; //窗口号、属性值和模式
    } simpleData;

    char *data;
};
```

4.6.2 容户与服务器间的接收事件处理

Qt 应用程序的 main 函数调用函数 exec()后进入事件主循环，事件主循环会循环调用 QEventLoop::processEvents()函数。来自客户的命令和服务器的事件经协议编码进入 socket 中，processEvents()函数从客户/服务器的 socket 描述符 csocket （对于多进程）或从通信队列（对于单进程）中取出命令或事件。客户端对服务器发来的事件进行处理后在客户端内进行分发，服务器对来自客户的命令调用相应的操作函数进行处理。

QEventLoop::processEvents() 函数的调用层次如图 4-13 所示。

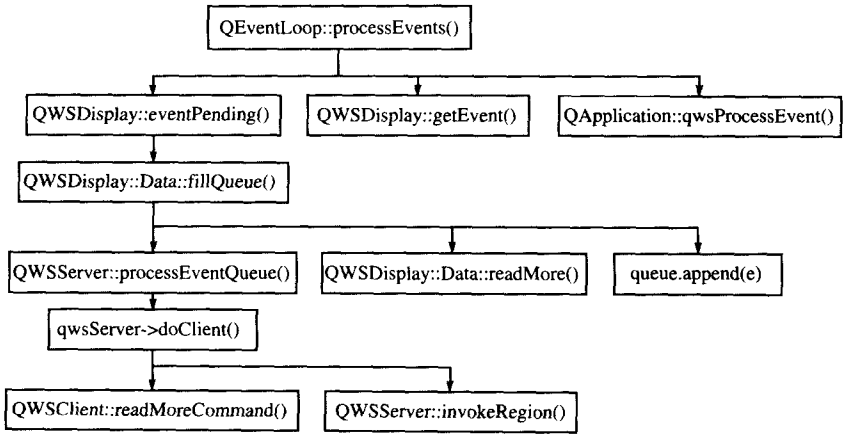


图 4-13　QEventLoop::processEvents()函数的调用层次

该函数列出如下：

```
bool QEventLoop::processEvents( ProcessEventsFlags flags )
{
    //处理来自 QWS server 事件
    int   nevents = 0;

#if defined(QT_THREAD_SUPPORT)
    QMutexLocker locker( QApplication::qt_mutex ); //多线程间用的互斥锁
#endif

    //处理 GUI 和邮递的事件
    //当应用程序类型为 QApplication::GuiServer，即为服务器时，qt_is_gui_used 为 FALSE
    if (qt_is_gui_used ) {  //处理客户端的来自 QWS 服务器事件
        QApplication::sendPostedEvents();
      //服务器处理事件，客户端从 socket 中得到事件并排队
        while ( qt_fbdpy->eventPending() ) {
            if ( d->shortcut )  {
                return FALSE;
            }

            QWSEvent *event = qt_fbdpy->getEvent();//    从队列中取出事件
            nevents++;
      //客户端把来自服务器的事件进行分发处理
            bool ret = qApp->qwsProcessEvent( event ) == 1;
            delete event;
            if ( ret )  {
                return TRUE;
            }
        }
    }
    extern QPtrQueue<QWSCommand> *qt_get_server_queue();
    if ( !qt_get_server_queue()->isEmpty() ) {
        QWSServer::processEventQueue(); // 服务器处理来自客户的命令
    }

    int nsel;
    do { //用于监控使用了 QSocketNotifier 对象的文件描述符是否有变化，如鼠标设备文件等

        nsel = select( highest + 1,                    // 总描述符数+1
                       &d->sn_vec[0].select_fds,    //读描述符
                       &d->sn_vec[1].select_fds,    //写描述符
                       &d->sn_vec[2].select_fds,    // 出错描述符
```

```
                    tm );                          //定时
    } while (nsel == -1 && (errno == EINTR || errno == EAGAIN));

    ……
    // 激活 socket notifiers
    if ( ! ( flags & ExcludeSocketNotifiers ) && nsel > 0 && d->sn_highest >= 0 )
    {
        // 如果 select 函数说明数据已在 socket 中准备好，那么它会设置 socket notifier 为挂起
        int i;
        for ( i=0; i<3; i++ ) {
            if ( ! d->sn_vec[i].list )
                continue;

            QPtrList<QSockNot> *list = d->sn_vec[i].list;
            QSockNot *sn = list->first();
            while ( sn ) {
                if ( FD_ISSET( sn->fd, &d->sn_vec[i].select_fds ) )  //有描述符被设置
                    setSocketNotifierPending( sn->obj );
            sn = list->next();
            }
        }

        nevents += activateSocketNotifiers();
    }
    …
}
```

函数 eventPending()从 socket 中得到事件，并判断事件队列是否为空，列出如下：

```
bool QWSDisplay::eventPending() const
{
#ifndef QT_NO_QWS_MULTIPROCESS
    d->flush(); //更新转储窗户端 socket 中数据
#endif
    d->fillQueue();//服务器处理事件，客户端对事件进行排队
    return d->queueNotEmpty();
}
```

函数 fillQueue()对于服务器来说，从 socket 中读出来自客户的命令并触发相应操作函数进行处理。对于客户端来说，把事件从 socket 中取出后放入队列，待以后处理。

函数 fillQueue()与鼠标事件相关的代码部分列出如下：

```
void QWSDisplay::Data::fillQueue()
{
    QWSServer::processEventQueue(); //如果是服务器，服务器处理来自客户的命令
    QWSEvent *e = readMore();       //从队列或 socket 中读出当前或下一个事件
    while ( e ) {
        if ( e->type == QWSEvent::Connected ) {//连接事件
            connected_event = (QWSConnectedEvent *)e;
            return;
        }
       ......
        } else if ( e->type == QWSEvent::Mouse ) {//鼠标事件
            if ( !qt_screen ) {
                delete e;
            } else {
                QWSMouseEvent *me = (QWSMouseEvent*)e;
                if ( mouseFilter )
                    mouseFilter(me); //鼠标事件过滤
                if ( mouse_event ) {
                    if ( (mouse_event->window() != e->window ()
                         || mouse_event->simpleData.state !-
                         me->simpleData.state )) {
```

```
                    queue.append( mouse_event );  //将事件加入队列中
                    mouse_event_count = 0;
                }
            ...
            }
        ...
    }
    ......
    e = readMore();//从队列或 socket 中读出当前或下一个事件
  }
}
```

函数 processEventQueue 在服务器中处理来自客户端的命令，列出如下：

```
void QWSServer::processEventQueue()
{
   if ( qwsServer ) //如果服务器存在
      qwsServer->doClient( qwsServer->client[-1] );  //处理来自客户端的事件
}

void QWSServer::doClient( QWSClient *client )
{
   QWSCommand* command=client->readMoreCommand();//从 socket 中读出一条命令

   while ( command ) {
      QWSCommandStruct *cs = new QWSCommandStruct( command, client );
      commandQueue.append( cs ); //放入命令队列
      // Try for some more...
      command=client->readMoreCommand();//从 socket 中读出一条命令
   }


   while ( !commandQueue.isEmpty() ) {
      commandQueue.first();
      QWSCommandStruct *cs = commandQueue.take();
      switch ( cs->command->type ) {
      ...
      case QWSCommand::Region:  //区域操作
         invokeRegion( (QWSRegionCommand*)cs->command, cs->client );
         break;
      ...
#ifndef QT_NO_QWS_CURSOR
      case QWSCommand::DefineCursor:  //定义光标
         invokeDefineCursor( (QWSDefineCursorCommand*)cs->command, cs->client );
         break;
      case QWSCommand::SelectCursor:  //选择光标
         invokeSelectCursor( (QWSSelectCursorCommand*)cs->command, cs->client );
         break;
#endif
      case QWSCommand::GrabMouse://捕捉鼠标，只有当前窗口部件对鼠标有响应
         invokeGrabMouse( (QWSGrabMouseCommand*)cs >command, cs->client );
         break;
      case QWSCommand::GrabKeyboard: //捕捉键盘，只有当前窗口部件对键盘有响应
         invokeGrabKeyboard( (QWSGrabKeyboardCommand*)cs->command, cs->client );
         break;
#if !defined( QT_NO_SOUND ) && !defined( Q_OS_MACX )
      case QWSCommand::PlaySound:  //播放声音
         invokePlaySound( (QWSPlaySoundCommand*)cs->command, cs->client );
         break;
#endif
#ifndef QT_NO_COP
      case QWSCommand::QCopRegisterChannel: //为应用程序间通信注册信道
         invokeRegisterChannel( (QWSQCopRegisterChannelCommand*)cs->command,
                                cs->client );
```

```
            break;
        case QWSCommand::QCopSend:  //在信道上发送信息
            invokeQCopSend( (QWSQCopSendCommand*)cs->command, cs->client );
            break;
#endif
        ......
        delete cs->command;
        delete cs;
    }
}
```

函数 qwsProcessEvent 处理并分发来自服务器的事件，下面列出了其中鼠标事件分发部分的代码：

```
int QApplication::qwsProcessEvent( QWSEvent* event )
{
    int oldstate = -1;
    bool isMove = FALSE;
    if ( event->type == QWSEvent::Mouse ) {//鼠标事件，从协议数据中分析出鼠标数据
        QWSMouseEvent::SimpleData &mouse = event->asMouse()->simpleData;
        isMove = mouse_x_root != mouse.x_root || mouse_y_root != mouse.y_root;
        oldstate = mouse_state;
        mouse_x_root = mouse.x_root;//x 轴坐标
        mouse_y_root = mouse.y_root;//y 轴坐标
        mouse_state = mouse.state;  //鼠标状态
    }

    if ( qwsEventFilter(event) )              //应用程序事件过滤
        return 1;

#ifndef QT_NO_QWS_PROPERTIES
    if ( event->type == QWSEvent::PropertyNotify ) {
        QWSPropertyNotifyEvent *e = (QWSPropertyNotifyEvent*)event;
        if ( e->simpleData.property == 424242 ) {      // 粘贴板
#ifndef QT_NO_CLIPBOARD
            if ( qt_clipboard ) {
                QCustomEvent e( QEvent::Clipboard, event );
                QApplication::sendEvent( qt_clipboard, &e );  //客户端分发事件
            }
#endif
        }
    }
    ......
#ifndef QT_NO_COP
    if ( event->type == QWSEvent::QCopMessage ) {//QCopChannel 消息客户端本地分发
        QWSQCopMessageEvent *e = (QWSQCopMessageEvent*)event;
        QCopChannel::sendLocally( e->channel, e->message, e->data );
        return 0;
    }
#endif
    ....

    switch ( event->type ) {

    case QWSEvent::Mouse: {              // 鼠标事件
        QWSMouseEvent *me = event->asMouse();
        QWSMouseEvent::SimpleData &mouse = me->simpleData;

        //  翻译一个 QWS 事件(QWSEvent)到分离的 move 和 press/release 事件 (QEvent)
        if ( isMove ) {
            QWSMouseEvent move = *me;
            move.simpleData.state = oldstate;
            widget->translateMouseEvent( &move, oldstate );
        }
        if ( (mouse.state&MouseButtonMask) != (oldstate&MouseButtonMask) ) {
```

```
            widget->translateMouseEvent( me, oldstate );
        }
        if ( qt_button_down && ( mouse_state & MouseButtonMask ) == 0 )
            qt_button_down = 0;

        break;
    }
    ......
    default:
        break;
    }

    return 0;
}
```

4.6.3 服务器对窗口的协调管理

每个 Qt 应用程序负责自己窗口系统的绘制及显示，但存在着应用程序之间的窗口协调管理的问题。例如不同应用程序的窗口获得焦点、不同应用程序之间的窗口哪个显示在桌面的最上面等。这些工作对于各个应用程序来说，都是不能实现的，而是由服务器统一协调管理。

在代表服务器的 QWSServer 类中，服务器通过使用私有的数据结构管理整个窗口系统数据，这些数据结构分别说明如下：

（1）QWSPropertyManager propertyManager：用于管理窗口属性数据。

（2）QPtrList<QWSMouseHandler> mousehandlers：用于管理鼠标。

（3）QPtrList<QWSKeyboardHandler> keyboardhandlers：用于管理键盘。

（4）QPtrList<QWSCommandStruct> commandQueue：用于存储和管理窗口发给服务器的命令。

（5）QPtrList<QWSWindow> windows：用于存储和管理所有的窗口。

（6）QMap<QString, QPtrList<QWSClient> > channels：用于管理 QCopChannel信道。

服务器协调管理窗口在函数 QWSServer::doClient()实现，下面仅分析它所调用的函数 QWSServer::invokeSetFocus。

函数 QWSServer::invokeSetFocus在服务器接收到 RequestFocus 命令时被调用，它用来为窗口设置或取消焦点。因为桌面上的所有窗口并不是同一个应用程序的窗口，因此在处理窗口焦点时，需要通过服务器来发事件到不同的客户应用程序来设置焦点，焦点的具体设置及绘制由对应的客户应用程序的窗口来完成。

函数 QWSServer::invokeSetFocus 的调用层次如图 4-14 所示。

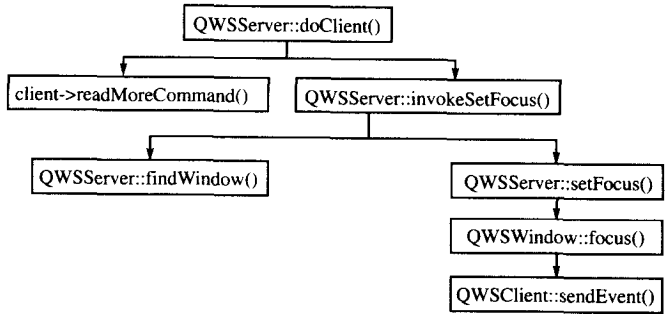


图 4-14 函数 QWSServer::invokeSetFocus 的调用层次

函数 QWSServer::invokeSetFocus 分析如下：

```
void QWSServer::invokeSetFocus( const QWSRequestFocusCommand *cmd, QWSClient *client )
{
    int winId = cmd->simpleData.windowid;
    int gain = cmd->simpleData.flag;//是否得到焦点的标识，0：失去焦点，1：得到焦点
   …
    QWSWindow* changingw = findWindow(winId, 0); //根据 winId 查找窗口
    if ( !changingw )
        return;

    if ( !changingw->forClient(client) ) {//窗口与窗口的客户端 client 不匹配
      qWarning("Disabled: clients changing other client's focus");
        return;
    }

    setFocus(changingw, gain);
}
```

函数 setFocus 通过发事件给客户端窗口来设置或取消焦点，分析如下：

```
void QWSServer::setFocus( QWSWindow* changingw, bool gain )
{
    if ( gain ) {// 设置焦点
        if ( focusw != changingw ) {//焦点所在窗口不是将要设置焦点的窗口
            if ( focusw ) focusw->focus(0);//如果焦点所在窗口存在，让它失去焦点
            focusw = changingw; //把 changingw 设为焦点所在窗口
            focusw->focus(1); // 设置焦点
            emit windowEvent( focusw, Active ); //发送焦点激活窗口事件
        }
    } else if ( focusw == changingw ) {//如果 changingw 是焦点所在窗口
        changingw->focus(0); // 使窗口失去焦点
        focusw = 0;
        //传递焦点给最近得到焦点的窗口
        QWSWindow* bestw=0;
        for (uint i=0; i<windows.count(); i++) {//查询窗口数组
            QWSWindow* w = windows.at(i);
            if ( w != changingw && !w->hidden() &&
                    (!bestw || bestw->focusPriority() < w->focusPriority()) )
                bestw = w;//得到最近有过焦点的窗口
            }
      //如果 bestw 不存在且窗口焦点时间计数存在，则还是原窗口得到焦点
        if ( !bestw && changingw->focusPriority() ) {
            bestw = changingw; // 必须是仅有的一个得到焦点的窗口
        }
        focusw = bestw;
        if ( focusw )
            focusw->focus(1); //设置焦点
        emit windowEvent( focusw, Active ); //发送焦点激活窗口事件
    }
}
```

函数 focus 通过发事件给对应的客户应用程序窗口来设置或取消焦点，列出如下：

```
void QWSWindow::focus(bool get)
 {
    if ( get )
        last_focus_time = global_focus_time_counter++;//保存上一次得到焦点时间计数
    QWSFocusEvent event;
    event.simpleData.window = id;
    event.simpleData.get_focus = get;
    c->sendEvent( &event );
}
```

4.7 输入设备与应用程序通信

系统中有指针设备（如鼠标和触摸屏等）及键盘设备用来接收输入操作。当这些设备的驱动程序模块安装到 Linux 操作系统后，指针设备的驱动程序设备文件类似于/dev/psaux 或/dev/ttyS0，键盘设备驱动程序设备文件为 /dev/tty1 或/dev/buttons。应用程序打开设备文件后，就可以从设备文件中读取数据，如鼠标的位置和键盘的输入字符等。

Qt 应用程序如何知道指针设备、键盘设备有数据输入，以及窗口系统发出相应事件？由于鼠标是指针设备的典型代表，而 Qt/Embedded 对键盘与鼠标设备的输入处理方法是相似的，甚至二者的差异大多只是函数名的不同而已，因此下面以一种 PC 机鼠标为例进行详细分析。

当键盘被击键或鼠标被点击时，键盘或鼠标设备产生中断，进入中断处理程序。在中断处理程序中，键盘或鼠标的动作被翻译成相应的数据存在一个循环缓冲区中。当应用程序需要输入数据时，则通过设备文件从这个循环缓冲区中读出数据。

当 Qt 应用程序作为 GuiServer 应用程序运行时，Qt 应用程序的 main 函数中的 QApplication 类在构造函数中会调用函数 QWSServer::startup(flags)来创建并启动 qwsServer 服务器，该服务器的启动函数 startup()会打开鼠标与键盘设备。通过 QSocketNotifier 类，利用信号与槽机制将打开的设备文件描述符 fd 与读取处理函数连接起来，通过信号 activated(int)来激活这个处理函数。

在 Qt 应用程序的 main 函数中调用 exec()函数时，应用程序进入主事件循环中。在这个循环中的事件处理函数 QEventLoop::processEvents 调用函数 select 来查询文件描述符 fd 是否有状态变化，如果有，将会调用函数 QEventLoop::activateSocketNotifiers 发出事件 QEvent::SockAct 。在 QSocketNotifier 类中重载了函数 event，它过滤出事件 QEvent::SockAct 后，发送信号 activated(sockfd)。

对于键盘和鼠标设备来说，上述过程中函数 select 查询的是键盘和鼠标设备文件描述符的状态变化。如果状态变化，说明设备有数据输入，发出信号。这样，应用程序即可处理鼠标或键盘的输入。

输入硬件设备与应用程序间通信原理如图 4-15 所示。在服务器与客户端，服务与输入硬件设备文件描述符之间通过 socket 的通信。而在服务器内部，输入设备信息以 QWSEvent 事件对象（如鼠标和键盘事件等）形式进行传递；在客户端内部，输入设备信息以 QEvent 事件对象（如鼠标和键盘事件等）形式进行传递。服务器打开输入设备后，将设备文件文件描述符与 socket 连接起来，然后通过 socket 接收输入设备的数据并经过过滤处理后发送给客户。

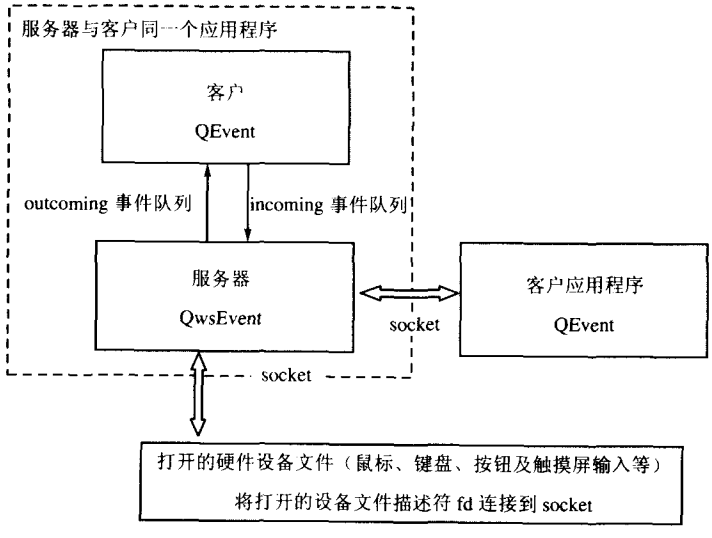


图 4-15 输入硬件设备与应用程序间的通信原理

4.7.1 鼠标键盘相关类

可以从 QWSKeyboardHandler 继承得到特定键盘的子类，这个类处理键盘／按钮的输入。QWSKeyboardHandler 读取/dev/tty，它代表了一个任意的底层 USB 事件设备（对于 USB 键盘来说）和一些 PDA 按钮设备的驱动程序。修改 QWSKeyboardHandler 允许支持不同类型的键盘，从其派生类来处理非指针输入设备是个更好的方法。

QWSKeyboardHandler 类在其构造函数中打开系统设备，在打开的设备上创建一个 QsocketNotifier。当 QSocketNotifier 收到数据，则调用 processKeyEvent()发送事件到 Qt/Embedded 重处理并发送到客户端。

可以从 QWSMouseHandler->QCalibratedMouseHandler 继承得到特定鼠标类型的子类，这个子类可以用于输入的鼠标/触摸屏的驱动程序。从 QCalibratedMouseHandler 继承的类使用过滤代码处理触摸屏上的指针，一些嵌入设备可以在内核中执行这种过滤操作，这种情况下驱动程序不必从 QCalibratedMouseHandler 继承。从 QCalibratedMouseHandler 派生的类适合于没有内核过滤的触摸屏，从 QWSMouseHandler 继承的类是增加其他类型指针设备的方法（如触摸屏和鼠标等）。

在 Qt/Embedded 中，QWSMouseHandler 类代表了鼠标驱动程序，它从系统设备处理事件并产生鼠标事件。QWSMouseHandler 类在其构造函数中打开设备，在打开的设备上创建 QSocketNotifier 对象。当 QSocketNotifier 对象收到数据时，将调用函数 mouseChanged()发送事件到 Qt/Embedded 重处理并发送到客户端。

4.7.2 打开鼠标设备的过程

在 qwsSever 服务器的构造过程中，它分别调用函数 QWSServer::openMouse 打开鼠标设备，调用函数 QWSServer::openKeyboard() 打开键盘设备。由于两个设备的打开过程很相似，所以下面只分析鼠标设备的打开过程。函数 QWSServer::openKeyboard()的调用层次如图 4-16 所示。

函数 openMouse 打开鼠标设备，从环境变量中得到鼠标设备文件的路径。它调用函数 open 打开设备文件，并通过 QSocketNotifier 将设备文件描述符与相应的处理函数连接起来。

函数 openMouse 列出如下：

```
void QWSServer::openMouse()
{
  //得到指针设备路径相关字符串
   QString mice = getenv("QWS_MOUSE_PROTO");
   if ( mice.isEmpty() ) {
#if defined(QT_QWS_CASSIOPEIA)
            mice = "Tpanel:/dev/tpanel";
#endif
           if ( mice.isEmpty() )
                mice = defaultMouse;  // "Auto" 或 "/dev/tpanel"
   }
   closeMouse(); //关闭指针设备
   bool needviscurs = TRUE;
   if ( mice != "None" ) {
#ifndef QT_NO_STRINGLIST
//将字符串以空格为界分成多个字符串
    QstringList mouse = QstringList::split(" ",mice);
    for (QstringList::Iterator m=mouse.begin(); m!=mouse.end(); ++m) {
       Qstring ms = *m;
#else
       Qstring ms = mice; // 假定仅有一个鼠标
       {
#endif
```

```
            QWSMouseHandler* h = newMouseHandler(ms);
            /* XXX handle mouse cursor visibility sensibly
              if ( !h->inherits("QcalibratedMouseHandler") )
              needviscurs = TRUE;
             */
          }
    }
#ifndef QT_NO_QWS_CURSOR
    setCursorVisible( needviscurs );
#endif
}
```

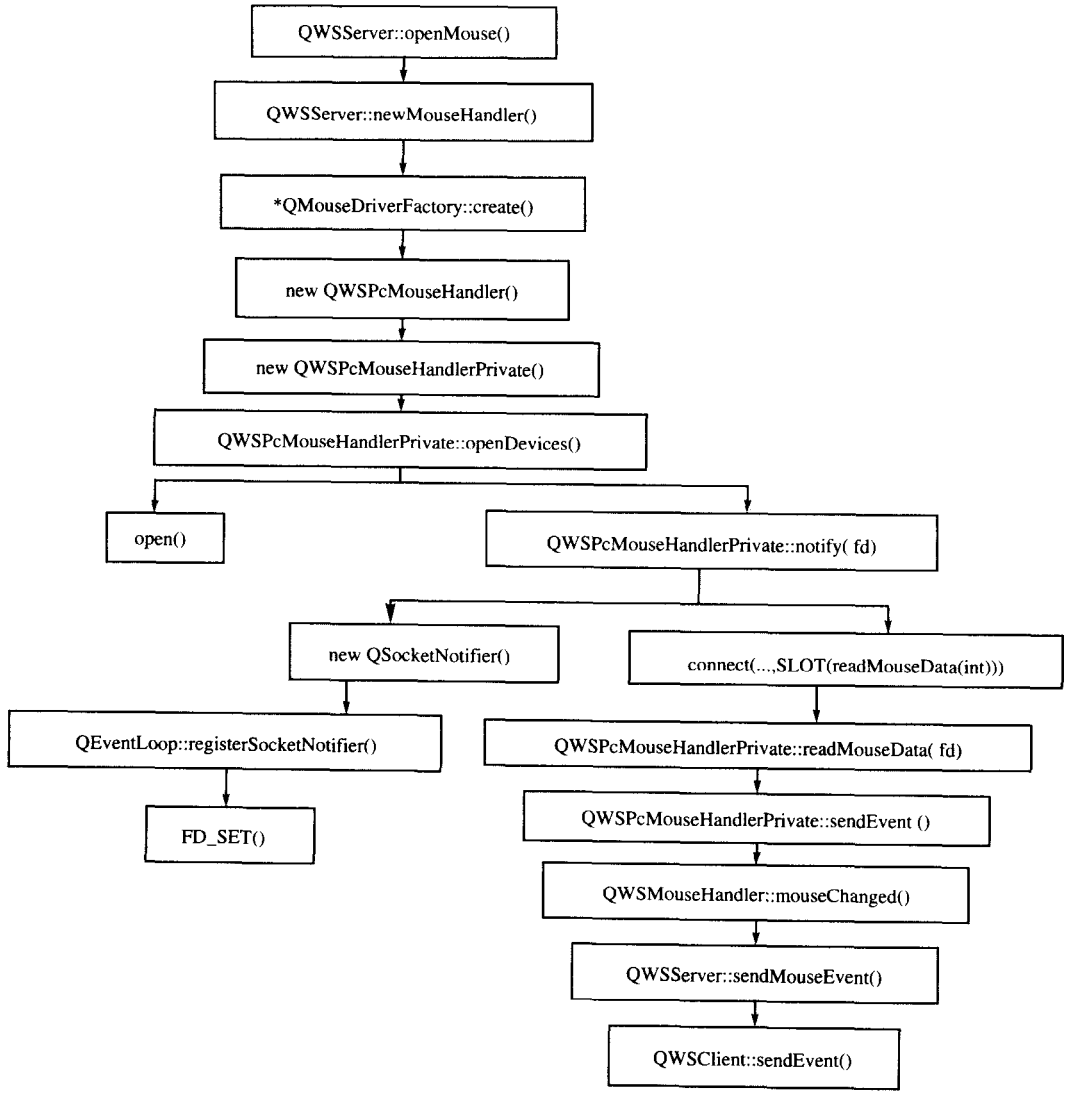


图 4-16　函数 QWSServer::openMouse()　的调用层次

函数 newMouseHandler 从环境变量字符串中分析出设备文件路径和鼠标协议名，然后调用函数 create 创建 QWSMouseHandler 对象，列出如下：

```
QWSMouseHandler* QWSServer::newMouseHandler(const QString& spec)
{
    static int init=0;
    if ( !init && qt_screen ) {
```

```
        init = 1;
    }

    int c = spec.find(':');
    QString mouseProto;
    QString mouseDev;
    if ( c >= 0 ) {
        //从诸如“TPanel:/dev/tpanel”字符串中得到协议TPanel
        mouseProto = spec.left(c);
        // 从诸如“TPanel:/dev/tpanel”字符串中得到设备名/dev/tpanel
        mouseDev = spec.mid(c+1);
    } else {
        mouseProto = spec;  //字符串中只有指针设备协议
    }

    QWSMouseHandler *handler = 0;
    handler = QMouseDriverFactory::create(mouseProto, mouseDev);
    return handler;
}
```

QMouseDriverFactory 类为 Qt/Embedded 创建 QWSMouseHandler 对象。鼠标驱动程序工厂使用 QMouseDriverFactory::create(key) 为一个指定的 key 创建一个 QWSMouseHandler 对象。驱动程序是内在或从一个驱动程序插件（QMouseDriverPlugin）动态装载的，这个类仅在 Qt/Embedded 中使用。QMouseDriverFactory::keys() 返回一个有效 key 的链表。

函数 QMouseDriverFactory::create 中仅列出了一种鼠标（PC 鼠标）的处理代码，后面的函数都是这种鼠标的处理代码。该函数列出如下：

```
QWSMouseHandler *QMouseDriverFactory::create( const QString& key,
                                    const QString &device )
{
    QString driver = key.lower();
......
#ifndef QT_NO_QWS_MOUSE_PC
    if ( driver == "auto" || driver == "intellimouse" ||
         driver == "microsoft" || driver == "mousesystems" ||
         driver == "mouseman" || driver.isEmpty() ) {
         qDebug( "Creating mouse: %s", key.latin1() );
         return new QWSPcMouseHandler( key, device );
    }
#endif
......
    return 0;
}

QWSPcMouseHandler::QWSPcMouseHandler( const QString &driver, const QString &device )
{
    d = new QWSPcMouseHandlerPrivate( this, driver, device );
}

QWSPcMouseHandlerPrivate::QWSPcMouseHandlerPrivate( QWSPcMouseHandler *h,
    const QString &drv, const QString &dev )
    : handler( h ), driver( drv ), device( dev )
{
    notifiers.setAutoDelete( TRUE );
    retries = 0;
    openDevices();
}
```

函数 openDevices 打开鼠标设备文件后测试设备，并调用函数 notify 将文件描述符与处理函数连接起来。该函数列出如下：

```
void QWSPcMouseHandlerPrivate::openDevices()
{
   nsub=0;
   int fd = -1;

   if ( !driver.isEmpty() && driver != "Auto" ) {
     // 指定特定的鼠标
     if ( driver == "IntelliMouse" ) {
        QString dev = device.isEmpty() ? Qstring("/dev/psaux") : device;
        fd = open( dev.latin1(), O_RDWR | O_NDELAY );  //打开设备文件
        if ( fd >= 0 )  //测试设备并存储 fd 及测试结果数据
          sub[nsub++] = new QWSPcMouseSubHandler_intellimouse(fd);
     }
      ......
     if ( fd >= 0 )
        notify(fd);
   }
   ......
}
```

4.7.3 设备文件与 socket 连接

函数 notify 为鼠标设备文件描述符 fd 创建一个 QSocketNotifier 对象，然后通过信号与槽机制将 fd 与读设备数据函数 readMouseData 连接起来。QSocketNotifier 类构造函数调用函数 registerSocketNotifier（this）这个函数调用 FD_SET()将 socked 描述符 socked 加入到文件描述符数组 fds 中，以便函数 select() 来查找 fds 并发出信号 activated(int)。一旦收到信号 activated(int)，就会触发函数 readMouseData 从鼠标设备中读取数据。

函数 notify 列出如下：

```
void QWSPcMouseHandlerPrivate::notify(int fd)
{
   QSocketNotifier *mouseNotifier
     = new QsocketNotifier( fd, QsocketNotifier::Read, this );
   connect(mouseNotifier, SIGNAL(activated(int)),this, SLOT(readMouseData(int)));
   notifiers.append( mouseNotifier );
}
```

类构造函数 QSocketNotifier() 注册 socket，将参数 type 类型的事件与 socket 连接起来，通常应该通过其成员函数 setEnabled() 明确地激活或关闭 QSocketNotifier。

类构造函数 QSocketNotifier() 列出如下：

```
QSocketNotifier::QSocketNotifier( int socket, Type type, QObject *parent,
                    const char *name )
   : QObject( parent, name )
{
   ...
   sockfd = socket;
   sntype = type;
   snenabled = TRUE;
   QApplication::eventLoop()->registerSocketNotifier( this );
}
```

函数 registerSocketNotifier 将 QSocketNotifier 对象指针和文件描述符存入结构中，并放入到挂起队列中。同时将文件描述符加入到描述符数组 d->sn_vec[type].enabled_fds 中，供 select 函数查询用。这样，文件描述符与 socket 连接起来了。

函数 registerSocketNotifier 列出如下：

```
void QEventLoop::registerSocketNotifier( QSocketNotifier *notifier )
{
   int sockfd = notifier->socket();
   int type = notifier->type();
   ...

   QPtrList<QSockNot>  *list = d->sn_vec[type].list;
   fd_set *fds  = &d->sn_vec[type].enabled_fds;
   QSockNot *sn;

   if ( ! list ) {//list 链表不存在
     // 创建一个新的 list 链表, QSockNotType 类析构函数会释放它
     list = new QPtrList<QSockNot>;
     Q_CHECK_PTR( list );
     list->setAutoDelete( TRUE );
     d->sn_vec[type].list = list;
   }

//创建结构 QSockNot 实例，并初始化其成员
sn = new QSockNot;
Q_CHECK_PTR( sn ) ;
sn->obj = notifier; // QSocketNotifier 对象指针
sn->fd = sockfd; //在鼠标设备打开操作中是鼠标设备文件描述符
sn->queue = &d->sn_vec[type].pending_fds; //挂起的文件描述符队列

if ( list->isEmpty() ) { //链表是空
     list->insert( 0, sn );
} else {                 //    链表按 fd 降序排列
     QSockNot *p = list->first();
     while ( p && p->fd > sockfd )
        p = list->next();
...
     if ( p )
        list->insert( list->at(), sn );
     else
        list->append( sn );
   }

   FD_SET( sockfd, fds ); //将文件描述符加入到描述符数组中，以便 select 函数查询
   d->sn_highest = QMAX( d->sn_highest, sockfd ); //比较优先级
}
```

函数 readMouseData 读取鼠标设备的数据，经处理后调用函数 sendEvent 发送事件给客户，列出如下：

```
void QWSPcMouseHandlerPrivate::readMouseData(int fd)
{
    for (;;) {
     uchar buf[8];
     int n = read(fd, buf, 8); //从鼠标设备中读取 8 个字节数据到 buf
     if ( n<=0 )
        break;
     for (int i=0; i<nsub; i++) {
         QWSPcMouseSubHandler& h = *sub[i];
         if ( h.file() == fd ) {
              h.appendData(buf,n); //  存储 buf数据到 h 对象的内部 buffer 中
              for (;;) {
                  switch ( h.useData() ) {//   分析数据,得到鼠标状态
                    case QWSPcMouseSubHandler::Button:
                     sendEvent(h); //处理并发送事件
                      break;
                case QWSPcMouseSubHandler::Insufficient :
                     goto breakbreak;
                case QWSPcMouseSubHandler::Motion:
```

```
                    break;
                }
            }
            breakbreak:
                ;
        }
    }
}
        //省略了鼠标可靠性检测代码
}
```

4.7.4 服务器捕获设备输入信号

在 Qt 应用程序 main 函数的主事件循环中，调用函数 select() 来检查文件描述符的变化。当文件描述符有变化时，主事件循环将调用 activateSocketNotifiers() 函数产生事件 QEvent::SockAct。而 QSocketNotifier 重载了 event 函数，它会接收过滤出事件 QEvent::SockAct 并发出信号 activated(sockfd)。这样，服务器绑定在这个信号上的函数 readMouseData() 将被激活来读取输入设备文件中的数据，经处理后发送给客户。

函数 QEventLoop::activateSocketNotifiers 的调用层次如图 4-17 所示。

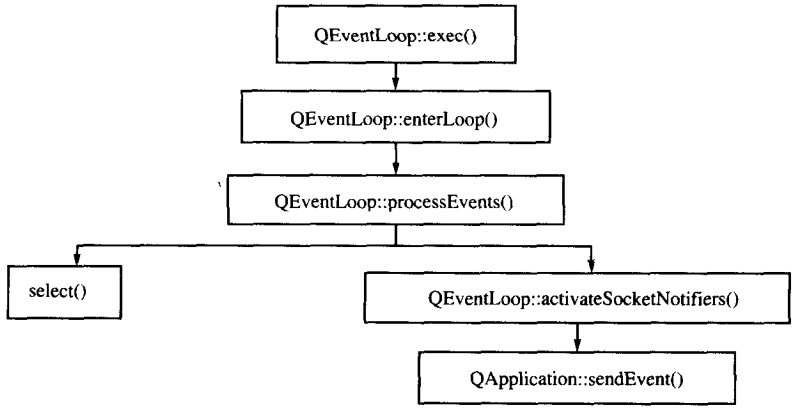


图 4-17 函数 QEventLoop::activateSocketNotifiers 的调用层次

函数 activateSocketNotifiers()查询有状态变化的描述符，并发出 SockAct，列出如下：

```
int QEventLoop::activateSocketNotifiers()
{
    if (d->sn_pending_list.isEmpty() )
        return 0;

    // activate entries
    int n_act = 0;
    QEvent event( QEvent::SockAct );  //构造事件 SockAct
    QPtrListIterator<QSockNot> it( d->sn_pending_list );
    QSockNot *sn;
    while ( (sn=it.current()) ) {//遍历有状态变化的描述符链表
      ++it;
      d->sn_pending_list.removeRef( sn );
      if ( FD_ISSET(sn->fd, sn->queue) ) {// sn->fd     在描述符数组 sn->queue中
          FD_CLR( sn->fd, sn->queue );  //从描述符数组 sn->queue 中清除这个描述符
          QApplication::sendEvent( sn->obj, &event ); //立即发送事件 SockAct
          n_act++;
      }
    }
```

```
    return n_act;
}
```

QSocketNotifier 重载 event 函数，重载函数 event 过滤出 SockAct 事件并发出信号 activated，这样对 activated 信号感兴趣的窗口部件可通过连接的槽函数做出相应的处理。重载函数 event 列出如下：

```
bool QSocketNotifier::event( QEvent *e )
{
    // Emits the activated() signal when a \c QEvent::SockAct is
    // received.
    QObject::event( e );          // will activate filters
    if ( e->type() == QEvent::SockAct ) {
      emit activated( sockfd );
      return TRUE;
    }
    return FALSE;
}
```

4.7.5 服务器发送事件给客户

服务器使用函数 sendEvent 函数将服务器事件通过 csocket 发送给客户端，并且对鼠标数据进行分析得到鼠标在屏幕上 的新位置，然后调用函数 mouseChanged 对鼠标位置与状态的变化进行处理。函数 sendEvent 列出如下：

```
bool QWSPcMouseHandlerPrivate::sendEvent(QWSPcMouseSubHandler& h)
{
    static const int accel_limit = 5;
    static const int accel = 2;

    if ( h.reliable() ) {//鼠标数据可靠
      QPoint motion = h.takeMotion(); //移动的相对位置
      if ( QABS(motion.x()) > accel_limit || QABS(motion.y()) > accel_limit )
         motion *= accel;
      QPoint newPos = handler->pos() + motion; //计算光标在屏幕上的绝对位置
      handler->limitToScreen( newPos ); //光标位置超出屏幕时进行限制处理

      handler->mouseChanged(newPos,h.buttonState());
      return TRUE;
    } else {//鼠标数据不可靠
      h.takeMotion();
      if ( h.buttonState() & (Qt::RightButton|Qt::MidButton) ) {
         // 对于用户来说，按下鼠标右键或中键没有移移动光标是件奇怪的事
         h.worse();//鼠标出错
      }
      return FALSE;
    }
}
```

当鼠标事件发生时，调用函数 mouseChanged 来处理，它调用函数 sendMouseEvent 在 qwsServer 服务器中发送鼠标事件。其参数 pos 表示鼠标的位置，参数 bstate 表示鼠标的按键的状态，即指示按下的按键。

函数 mouseChanged 列出如下：

```
void QWSMouseHandler::mouseChanged( const QPoint& pos, int bstate )
{
    mousePos = pos;
    QWSServer::sendMouseEvent(pos,bstate);
}
```

函数 sendMouseEvent 是 Qt 内部使用的一个函数，它在 qwsServer 服务器中发送鼠标事件。qwsServer 服务器处理鼠标在屏幕上的显示（如鼠标箭头处理），并发送事件。

函数 sendMouseEvent 列出如下：

```
void QWSServer::sendMouseEvent(const QPoint& pos, int state)
{
    const int btnMask = LeftButton | RightButton | MidButton;
    qwsServer->showCursor(); //显示光标

    if ( state )
      qwsServer->screenSaverWake();//唤醒屏幕

    mousePosition = pos;
    qwsServer->d->mouseState = state;

    QWSMouseEvent event;

    //如果捕捉的窗口消失，捕捉还是激活的，直到鼠标释放为止
    QWSWindow *win = qwsServer->mouseGrabber ? qwsServer->mouseGrabber : qwsServer->windowAt( pos );
    event.simpleData.window = win ? win->id : 0;  //  得到窗口 ID

#ifndef QT_NO_QWS_CURSOR
    qt_screencursor->move(pos.x(),pos.y());  //  移动屏幕光标的位置

    // 桌面上的鼠标箭头处理，如果上次事件在个窗口上，prevWin 为 TRUE
    if (!win && prevWin) {
      if ( !qwsServer->mouseGrabber )//设置光标为箭头
         qwsServer->setCursor(QWSCursor::systemCursor(ArrowCursor));
      else
         qwsServer->nextCursor = QWSCursor::systemCursor(ArrowCursor);
      prevWin = FALSE;

    // reset prevWin
    if (win && !prevWin)
      prevWin = TRUE;
#endif

    if ( (state&btnMask) && !qwsServer->mouseGrabbing ) {
      qwsServer->mouseGrabber = win;
    }
    //设置事件event 数据
    event.simpleData.x_root=pos.x() ;
    event.simpleData.y_root=pos.y() ;
    event.simpleData.state=state | qws_keyModifiers;
    event.simpleData.time=qwsServer->timer.elapsed()  ;

    QWSClient *serverClient = qwsServer->client[-1];  //得到客户端
    QWSClient *winClient = win ? win->client() : 0;

    if ( serverClient )  //服务器的客户端
      serverClient->sendEvent( &event );  //发送事件给客户端
    if ( winClient && winClient != serverClient )
      winClient->sendEvent( &event );

    /*确信如果我们离开一个窗口，这个窗口得到最后一个鼠标事件以致于这个窗口知道鼠标已离开*/
    QWSClient *oldClient = qwsServer->d->cursorClient; //  得到光标所在的客户端
    ///光标所在的客户端不是服务器操作正在操作的客户端
    if ( oldClient && oldClient != winClient && oldClient != serverClient )
      oldClient->sendEvent( &event );

    qwsServer->d->cursorClient = winClient;

    if ( !(state&btnMask) && !qwsServer->mouseGrabbing )
```

```
    qwsServer->releaseMouse(qwsServer->mouseGrabber); //释放鼠标处理
}
```

函数 sendEvent 发送事件，列出如下：

```
void QWSClient::sendEvent( QWSEvent* event )
{
#ifndef QT_NO_QWS_MULTIPROCESS
    if ( csocket ) { //多进程应用程序
      if (csocket->state() == QSocket::Connection) {
         event->write( csocket ); //将事件数据写入到 socket 中
         csocket->flush();
      }
    }
    else
#endif
    {//单进程应用程序，服务器与客户在同一个应用程序
      qt_client_enqueue( event );//  将事件数据写入到队列中
    }
}
```

对于单进程应用程序，函数 qt_client_enqueue 将事件数据写入到全局队列中 incoming，列出如下：

```
static QPtrQueue<QWSEvent> incoming;
void qt_client_enqueue(const QWSEvent *event )
{
    QWSEvent *copy = QWSEvent::factory( event->type );
    copy->copyFrom( event );
    incoming.enqueue( copy );
}
```

4.8 应用程序显示

Qt 应用程序的窗口部件及光标等需要显示在显示屏上， Qt 应用程序中完成显示功能的部分称为“Display”（显示）。Display 的初始化由静态函数 init_display 完成。

4.8.1 与显示相关的类

与显示相关的类如下。

（1）QScreen 及其派生类

QScreen->QLinuxFbScreen->加速的屏幕显示 QTransformedScreen->QvfbScreen。

这些类包裹了 Qt/Embedded 绘制所用的帧缓冲，为旋转帧缓冲内容提供了坐标映射支持。它允许处理调色板，并且通过使用独立的帧缓冲内存。从而为设备提供屏幕画面外显存的访问，该显存被用于缓冲位图和允许加速的位图绘制。 QLinuxFbScreen 和加速的屏幕显示使用 Linux /dev/fb 接口访问显存和关于设备特征的信息，QWS_DISPLAY 定义被打开的帧缓冲设备，仅 QTransformedScreen 支持帧缓冲旋转。QVfbScreen 提供了一个包含一个仿真的帧缓冲的 X 窗口，仿真的帧缓冲是设置为“framebuffer”的共享内存，并使用位图绘制的 X 窗口。其目的是调试设备，以允许用户在 Qt/Embedded 没有离开 X 窗口情况下调试应用程序。加速的屏幕驱动程序检查它们是否能驱动 QWS_CARD_SLOT 定义的设备（QWS_CARD_SLOT 默认下是 AGP插槽的位置），并且从/dev/mm 中映射（mmap）芯片上的寄存器，并且还可做芯片特定的初始化工作（初始化寄存器到一个已知的值等）。最后， QScreen 用来创建新的 QXcreenCursors 和 QGfxes。如果需要修改分配在内存中的位图方式，则修改 QLinuxFbScreen 类，或从其派生类。如果编写一个加速驱动程序，则必须从 QScreen 或 QLinuxFbScreen 派生类。

（2）QScreenCursor 及其派生类

QScreenCursor->加速的 cursor->QvfbCursor，这些类处理屏幕上光标（cursor）的绘制，为非加速的 cursor 类型保护和恢复 cursor下面的屏幕显示。在加速的驱动程序中，从 QScreenCursor 中派生类是可选的（如果支持硬件 cursor，才会这样做）。

（3）QGfx 及其派生类

QGfx->RasterBase->Raster->加速的驱动程序 ->QGfxVfb->QgfxTransformedRaster,这个类包裹了绘制操作，类似低层次的 QPainter。QgfxRaster 及其派生类被用于将图形绘制到一个原始数据流的帧缓冲中，它们在帧缓冲中采用地址偏移和可裁剪的区域来进行窗口绘制。为了使用加速驱动程序，必须从 QGfxRaster派生类。通过这个派生类，可使绘制动作个性化，并加入新的 bit位的颜色深度/分辨率格式的支持。

（4）QLock 类

Qlock 和 QLockHolder类包裹了 System V 信号量，用于同步在 Qt/Embedded客户端之间共享内存的访问。QLockHolder 是一个用来管理和销毁 QLock 的实用类，没有必要修改此类。

（5）QDirectPainter类

QdirectPainter 类是一个 QPainter 类，用于提供一个 framebuffer 的指针。用其容易进行窗口内容在像素层次上的处理，不应该修改此类。

（6）QWSSoundServer 类

QWSSoundServer 类代表了声音播放服务器，它是 Qt/Embedded 服务器包含的一个简单声音播放器和混音器。客户端（Client）能请求服务器播放特定文件的声音，没有必要修改此类。

（7）QWSDisplay类

QWSDisplay 类仅存在 Qt/Embedded 服务器中，并且保持跟踪系统中的顶层窗口及键盘和鼠标。只有在需要大范围修改 Qt/Embedded 窗口行为，如 alpha 混合窗口时，才需要修改此类。

QWSDisplay::data类管理客户端的 QWS 客户端，从 QWSServer 中读取和解释事件。它在应用程序启动时连接到 QWS 服务器端，得到帧缓冲的信息，并创建内存管理器。在 QLinuxFbScreen 中，其他关于帧缓冲的信息直接来自/dev/fb。不应该修改这个类。

（8）QWSManager 类

QWSManager 类提供了对 Qt/Embedded 窗口管理，如处理窗口最大化、最小化，以及用户对移动和改变窗口尺寸等的请求。不应该修改它。

（9）QWSDecoraton 类

QWSDecoraton 类继承者是 Qt/Embedded 窗口管理器的不同风格，例如，QWSWindowsDecoration 以 Windows CE 风格绘制 Qt/Embedded 窗口框架。从其派生类是为了提供一个新的窗口管理器外观。

（10）QWSPropertyManager类

QWSPropertyManager 类提供了 QWS 客户端对 QWS 属性系统的接口（X 属性系统的一个简单版本，允许任意附加数据到顶层窗口，以一个整数为 key）。不应该修改此类。

（11）QWSRegionManager 类

QWSRegionManager类被客户端和服务器用来帮助管理顶层窗口区域，不应该修改此类。

4.8.2 与字体相关的类

Qt/Embedded 使用字体管理器 QFontManager 管理字体，与字体相关的类在客户端运行，下面分别说明这些类。

（1）QFontManager 类

每个应用程序有一个 QFontManager 用来管理字体。在应用程序启动时从$QTDIR/etc/fonts/fontdir（如果 QTDIR 未定义，则从/usr/local/etc/qt-embedded/fonts/fontdir）下读取字体定义文件。它还保持对

所有字体信息进行跟踪，并拥有使用字体的缓存 cache。它还使用字体工厂 QFontFactory 来增加新字体，字体工厂提供一个申请特定字体高层次的接口，并且调用 QFontFactories 函数从请求的硬盘上装载字体。注意，仅应用 BDF 和 Truetype 字体。Qt/Embedded 的优化.qpf 字体文件格式不使用 QFontManager 机制。没有必要修改这个类，除非希望改变字体匹配或 cache 行为。

（2）QDiskFont 类

QDiskFont 装有在硬盘上字体文件的信息（如/usr/local/etc/qt-embedded/tmes.ttf），包括文件路径、字体是否可缩放、字体的重量、大小及字体的 Qt/Embedded 名字等信息，这些信息被用来方便 QFontManager 在硬盘上找到更匹配的字体（使用一种加权的记分机制分配匹配字体的名字、是否斜体及字体的重量等）。这个类不应该被修改。

（3）QRenderedFont 类

QRenderedFont 类用于当前系统装载的一种字体，每个当前装载的字体仅有一个 QrenderedFont，它使用了引用计数。一旦没有程序使用 QRenderedFont，它及其字体位图将被删除。然而被装载字体的源 QDiskFont 还保持着被 QFontFactory 打开。不需要修改这个类。

（4）QFontFactory 类

QFontFactory 和派生类 QFontFactoryBDF，QFontFactoryTtf 一起提供了对特殊字体格式的支持。例如，可缩放的 Truetype 和 Type1 格式（在 QfontFactoryFtf 中被支持，它使用了 Freetype 2），并且位图 BDF 格式被 X 窗口使用。这个类用来打开打开在硬盘上的一种字体，一旦打开字体，为了从硬盘上字体快速创建新字体实例，它保持一直打开。该类还能创建一个 QRenderedFont,并能转换 Unicode 值到字体文件的序号一字的图形轮廓按次序（字体编码索引）存储在字体文件中，而不是 Unicode 值次序存储。不需要修改这个类。

（5）QGlyph 类

QGlyph 这个类描述了来自一个 QRenderedFont 字符的特定图像，例如特定字体的字母 A。它装有指向一个 QGlyphMetrics结构的指针，该结构存有字符的信息和字体轮廓的原始数据，即 1 位掩码和 8位 alpha 通道。一般情况下不需要修改此类。如果在支持浮雕字体等情况下需要修改此类，则还需要修改 QGfxRaster类。

QMemoryManagerPixmap/QMemeoryManager 类处理位图对空间的需要，还跟踪 QPF 格式字体(这些是 QRenderedFonts 较小的状态 dump，典型值是 220 KB。为了节约内存的使用，它们能直接从硬盘映射。如果发现一个 QPF 字体，它匹配的字体不需要创建 QRenderedFont。如果字体需求较少，如仅需要一个固定大小的字体，删除所有的 QFontFactory 支持，并简单地使用 QPF 是可能的。

4.8.3 函数 init_display

函数 init_display 初始化应用程序的显示部分 Display，它打开帧缓冲设备，创建代表帧缓冲设备的 QWSDisplay 实例 qt_fbdpy。并且通过 qt_fbdpy，应用程序可以访问帧缓冲设备。函数 init_display 还初始化颜色、字体及光标相关的类，建立全局对象或数据结构。

函数 init_display（在 src/kernel/qapplication_qws.cpp 中）列出如下：

```
static void init_display()
{
    if ( qt_fbdpy ) return; //如果 Display 存在，返回

    setlocale( LC_ALL, "" );        //  使用正确的字符集映射
    setlocale( LC_NUMERIC, "C" ); //    使 sprintf()/scanf()工作

    //打开帧缓冲设备，设置调色板
    qt_fbdpy = new QWSDisplay();

    QColor::initialize(); // 初始化颜色模式,如 8位等
```

```
    QFont::initialize(); //    创建全局的字体缓存
#ifndef QT_NO_CURSOR
    QCursor::initialize(); // 初始化光标形状，并存在光标表中
#endif
    QPainter::initialize(); //    目前为空
#if defined(QT_THREAD_SUPPORT)
     QThread::initialize(); // 创建线程池
#endif
    QFontManager::initialize(); //创建对象实例，存在全局变量 qt_fontmanager 中
#ifndef QT_NO_QWS_MANAGER
    /*QWSDecoration 类允许 Qt/Embedded 窗口管理器 (Window Manager) 个性化地设置外观，Qt/Embedded 对顶层窗口提供了管
理。通过从 QWSDecoration 派生类，并重载一些方法函数，窗口管理器可以管理窗口的的边框和按钮能被个性化*/
    qws_decoration = QWSManager::newDefaultDecoration() ;
#endif

    incoming.setAutoDelete(true);
    outgoing.setAutoDelete(true);

    qApp->setName( appName ); //设置应用程序的名字

    QFont f;
    f = QFont( "helvetica", 10 );
    QApplication::setFont( f ); //设置字体

    qt_set_qws_resources(); //设置调色板颜色
}
```

4.8.4 QWSDisplay 构造函数分析

类 QWSDisplay 构造函数创建一个类对象，它管理图形设备的帧缓冲驱动程序，包括显存的分配，以及调色板和光标的管理等。它是一个 Qt/Embedded 内部使用的类，其调用层次如图 4-18所示。

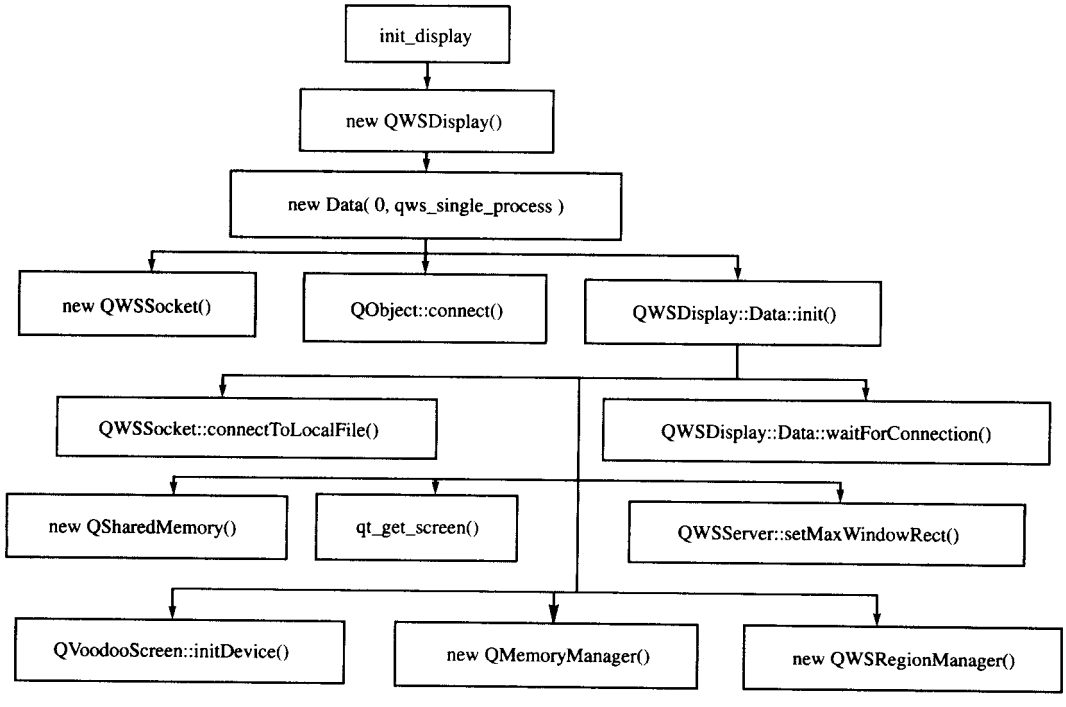


图 4-18 构造函数 QWSDisplay() 的调用层次

QWSDisplay 类构造函数列出如下：

```
QWSDisplay::QWSDisplay()
{
    d = new Data( 0, qws_single_process ); // qws_single_process 表示是否是单线程
}
```

QWSDisplay::Data 在应用程序启动时连接到 QWS 服务器端得到帧缓冲的信息，创建或得到共享内存，并创建内存管理器。若是服务器，初始化帧缓冲设备。在 QLinuxFbScreen 中，其他关于帧缓冲的信息直接来自/dev/fb。

类 QWSDisplay::Data 列出如下：

```
class QWSDisplay::Data
{
public:
    Data( QObject* parent, bool singleProcess = FALSE )
    {
#ifndef QT_NO_QWS_MULTIPROCESS
        if ( singleProcess ) //单线程情况，服务器是单线程
            csocket = 0;
        else {  //多线程及客户端
            csocket = new QWSSocket(parent); //创建 socket
            QObject::connect( csocket, SIGNAL(connectionClosed()),
                    qApp, SLOT(quit()) );
        }
#endif
        init();
    }
    ......
//公共数据成员
    //QWSRegionManager 被客户端和服务器用来帮助管理顶层窗口区域
    QWSRegionManager *rgnMan;
    uchar *sharedRam;
#if !defined(Q_NO_QSHM) && !defined(QT_NO_QWS_MULTIPROCESS)
    QSharedMemory shm;
#endif
   ......
}
```

如果应用程序包含服务器，函数 init 创建用于显示与绘图的共享内存并映射到虚拟地址空间。在需要显示时，拷贝到帧缓存初始化帧缓冲设备。如果应用程序是客户端，函数 init 得到可共享内存，并映射到虚拟地址空间。对于所有应用程序，函数 init 打开帧缓冲设备得到代表帧缓冲设备的 QScreen 对象、设置最大窗口尺寸、初始化光标、创建 QMemeoryManager 对象、处理位图对空间的需要、创建 QWSRegionManger 对象并且管理顶层窗口区域。

函数 init 列出如下：

```
void QWSDisplay::Data::init()
{
    connected_event = 0;
    region_ack = 0;
    mouse_event = 0;
    region_event = 0;
    region_offset_window = 0;
#ifndef QT_NO_COP
    qcop_response = 0;
#endif
    current_event = 0;
    mouse_event_count = 0;
    ramid = -1;
    mouseFilter = 0;
/*得到用于 server/client 通信的文件名，类似于“/tmp/qtembedded-unknown/ QtEmbedded-0” */
```

```
    QString pipe = qws_qtePipeFilename();

    sharedRamSize = qwsSharedRamSize; // 100 * 1024

#ifndef QT_NO_QWS_MULTIPROCESS
    if ( csocket )
    {// QWS client
        csocket->connectToLocalFile(pipe); //创建本地的 socket，连接 socket 到文件 pipe
      QWSIdentifyCommand cmd;
     //将 appName 转换到 Qt/Embedded 的 server/client 通信数据格式
      cmd.setId(appName);
#ifndef QT_NO_QWS_MULTIPROCESS  //多线程
      if ( csocket )
         cmd.write( csocket ); //将原始数据流在 socket 上发送
      else
#endif
        //在服务器的全局命令队列 outgoing 上排队
         qt_server_enqueue( &cmd );

      //等待连接验证
      waitForConnection();

      qws_client_id = connected_event->simpleData.clientId;//得到客户端 ID

      // 如果我们没有定义任何内容，现在想得到精确的 display 定义
      if ( qws_display_spec[0] == ':' )
         qws_display_spec = connected_event->display;

      if ( !QWSDisplay::initLock( pipe, FALSE ) )//锁不存在
         qFatal( "Cannot get display lock" );

      //得到名为 pipe+ 'c' ，大小为 0 的共享内存文件名
       shm = QSharedMemory(0,pipe.latin1(),'m');
      //用于显示的共享内存由服务器创建，客户端得到共享内存并映射进本应用程序空间
      if (shm.create() && shm.attach()) {
         //打开帧缓冲设备，并建立画面外的显存管理及调色板
          QScreen *s = qt_get_screen( qws_display_id, qws_display_spec );
         sharedRamSize += s->memoryNeeded(qws_display_spec);
      } else {
         perror("Can't attach to main ram memory.");
         exit(1);
      }
      sharedRam = (uchar *)shm.base();  //共享内存的基地址
    } else
#endif
    {

      // QWS server
      if ( !QWSDisplay::initLock( pipe, TRUE ) )  //创建锁
         qFatal( "Cannot get display lock" );
     //得到 QScreen 对象，它代表了按特定图形设备及显示屏号建立的帧缓冲
      QScreen *s = qt_get_screen( qws_display_id, qws_display_spec );
      sharedRamSize += s->memoryNeeded(qws_display_spec);  //得到帧缓存大小

#ifndef QT_NO_QWS_MULTIPROCESS
      //得到名为 pipe+ 'c' ，大小为 sharedRamSize 的共享内存文件名
      shm = QSharedMemory(sharedRamSize,pipe.latin1(), 'm');
      if (!shm.create())  //创建 sharedRamSize 大小的共享内存
         perror("Cannot create main ram shared memory\n");
      if (!shm.attach())  //映射到应用程序的虚拟地址空间
         perror("Cannot attach to main ram shared memory\n");
      sharedRam = (uchar *)shm.base();
#else  //对于单进程，只需要分配一块内存即可，不需要多进程间的共享内存
      sharedRam=(uchar *)malloc(sharedRamSize);
#endif
```

```
        //共享内存清 0
        memset(sharedRam,0,sharedRamSize);

        QWSIdentifyCommand cmd;
        cmd.setId(appName);  //在服务器中存储应用程序名
        qt_server_enqueue( &cmd );
    }
    //设置最大窗口尺寸，并通过 QWSClient::sendMaxWindowRectEvent()发送事件
    setMaxWindowRect(QRect(0,0,qt_screen->width(),qt_screen->height()));
    int mouseoffset = 0;

    /*也允许一些内存给图形驱动程序用，注意 sharedRamSize()有副作用，它仅被在函数 initDevice()之前调用 1 次。这个工作类似
于 QScreenCursor 代码。sharedRam+sharedRamSize 指向共享结构的末尾，返回保留的内存量*/
    sharedRamSize -= qt_screen->sharedRamSize(sharedRam+sharedRamSize);
    #ifndef QT_NO_QWS_MULTIPROCESS
        if(!csocket)
    #endif
        {/*这个函数在 QWS 服务进程中调用，它关闭闪烁的控制台、建立调色板、激活帧缓冲上的写操作并且初始化画面外的显存管理*/
          qt_screen->initDevice();
        }

    #ifndef QT_NO_QWS_CURSOR
        //初始化光标
        mouseoffset=qt_screen->initCursor(sharedRam + sharedRamSize,!csocket );
    #endif

        sharedRamSize -= mouseoffset;
        sharedRamSize -= sizeof(int);
        qt_last_x = (int *)(sharedRam + sharedRamSize);
        sharedRamSize -= sizeof(int);
        qt_last_y = (int *)(sharedRam + sharedRamSize);

        /* QMemoryManagerPixmap/QMemeoryManager  类处理位图对空间的需要，还跟踪 QPF 格式字体。其构造函数初始化帧缓冲内存管
理器，加 4 KB来避免挤垮硬件光标。帧缓冲内存管理器管理未使用的显存，可用其来存储位图或字体.*/
        int screensize=qt_screen->screenSize()  ;
        memorymanager=new QMemoryManager(qt_screen->base()+screensize+4096,//未使用的显存起始地址
            qt_screen->totalSize()-(screensize+4096),//未使用的显存大小
            0);//没使用的 fontrom

    /*创建 QWSRegionManger 对象，如果 csocket 存在表示给客户端创建；否则为服务器创建。QWSRegionManager 被客户端和服务器用
来帮助管理项层窗口区域，管理区域和每区域矩形数据*/
    #ifndef QT_NO_QWS_MULTIPROCESS
        rgnMan = new QWSRegionManager( pipe, csocket );
    #else
        rgnMan = new QWSRegionManager( pipe, 0 ); //####### not necessary
    #endif
    #ifndef QT_NO_QWS_MULTIPROCESS
        if ( csocket )
          csocket->flush();
    #endif
    }
```

4.8.5 QScreen

目前高级的嵌入式 ARM 体系 CPU 大多集成了 LCD 控制模块，该模块通过 DMA 硬件自动传递数据。Framebuffer 驱动程序的实现分为两个方面，一方面是对 LCD 及其相关部分的初始化，包括创建缓冲区的画面和对 DMA 通道的设置；另外一方面是读写画面缓冲区，可通过 read、write 及 lseek 等系统调用接口进行。当设置 DMA 通道和画面缓冲区完成后，DMA 开始正常工作。硬件自动将画面缓冲区的内容输出到 LCD 显示屏上，不需要软件的干预。framebuffer 驱动程序必须将帧缓冲映射到用户空间，这样对于该画面映射空间的写操作即可实时地显示在 LCD 上。这里所说的驱动程序不是指硬

件驱动程序模块，而是应用程序对驱动程序模块功能的封装。

应用程序的显示都是通过 QScreen 类来操作的，在服务器中初始化显示设备。在每个客户端应用程序中打开显示设备，并执行窗口的显示操作。

QScreen 类和 QGfx 类是 Qt/Embedded 图形引擎的底层抽象，QScreen 类是抽象的底层显示设备基类，它声明了显示设备的基本描述和操作方式，如打开、关闭、显示及创建 QGfx 对象操作对象等；QGfx 类抽象了显示设备的图形设备环境，如画刷、画矩形及画线等操作。两个基类的函数基本上是虚函数，具体显示设备，如 Linux 的 framebuffer及 Qt Virtual Framebuffer 的抽象接口类将从这两个基类中继承并重载基类中的虚函数。

对于基本的 framebuffer 设备，Qt/Embedded 用 QLinuxFbScreen 处理。针对具体显示硬件（如 ATI 显示卡）的加速特征，Qt/Embedded 从 QLinuxFbScreen 和图形设备环境模板类 QGfxRaster<depth, type> 继承相应子类，并针对相应硬件重载相关虚函数。

类 QScreen 及其继承者管理 framebuffer 和调色板， QScreen 是屏幕光标和 QGfx 的工厂。QLinuxFbScreen 管理一个 Linux 的 framebuffer，加速的驱动程序子类 QLinuxFbScreen。一个 Qt/Embedded 应用程序仅能有一个屏幕。

函数 qt_get_screen 返回一个 QScreen 类的派生类，参数 display_id 是 Qt/Embedded 服务器连接到的显示屏号，spec 是驱动程序名（如 Mach64:/dev/fb0）. QGfxDriverFactory 用来查询合适的驱动程序，如果找到，创建一个 QScreen 派生类对象代表帧缓冲驱动程序。用户写新的图形驱动程序时应该创建一个 QScreen 派生类加到 QGfxDriverFactory 或使用 QGfxDriverPlugin 创建一个动态装载的驱动程序插件。

函数 qt_get_screen 还把图形设备（如显卡）的帧缓冲映射到应用程序的虚拟空间，这与上面所说的显示用共享内存不是一个地址空间。共享内存用来建立需要显示的图形，然后拷贝到帧缓冲映射中显示，即在帧缓冲映射地址空间的数据会立即显示在屏幕上。

函数 qt_get_screen 的调用层次如图 4-19 所示。

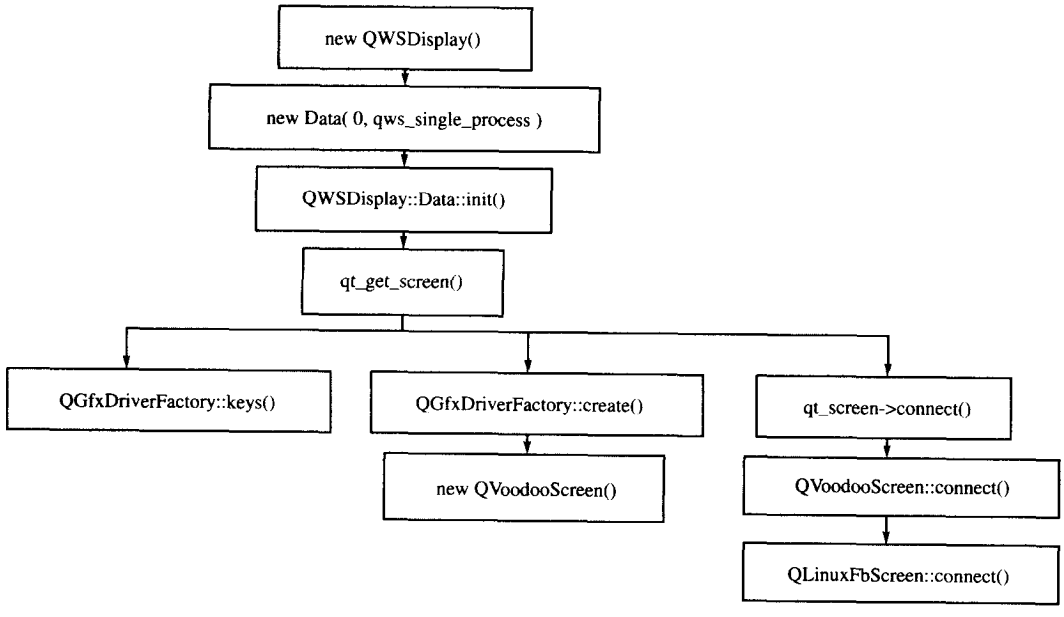


图 4-19 函数 qt_get_screen 调用层次

函数 qt_get_screen（在 src/kernel/qgfxraster_qws.cpp 中）分析如下：

```
QScreen *qt_get_screen( int display_id, const char *spec )
```

```
{
    QString displaySpec( spec );
    QString driver = displaySpec;
    int colon = displaySpec.find( ':' );
    if ( colon >= 0 )
      driver.truncate( colon );

    bool foundDriver = FALSE;
    QString driverName = driver;

#ifndef QT_NO_STRINGLIST
    //得到各种图形驱动程序名的字符串链表
    QStringList driverList = QGfxDriverFactory::keys();
    QStringList::Iterator it;
    for ( it = driverList.begin(); it != driverList.end(); ++it ) {
    //driver 为空或为匹配字符串链表中驱动程序名
      if ( driver.isEmpty() || QString( *it ) == driver ) {
        driverName = *it;
#endif
        qt_screen = QGfxDriverFactory::create( driverName, display_id );
        if ( qt_screen ) {
          foundDriver = TRUE;
          if ( qt_screen->connect( spec ) ) {  //打开帧缓冲设备，并映射到虚拟空间
            return qt_screen;
          } else {
            delete qt_screen;
            qt_screen = 0;
          }
        }
#ifndef QT_NO_STRINGLIST
      }
    }
#endif
   ...
    return 0;
}
```

Qt/Embedded 使用 QGfxDriverFactory 类创建 QScreen 对象，这个类是图形驱动程序工厂，它使用 QGfxDriverFactory::create(key)为给定的 key 创建一个 QScreen 对象。驱动程序可以内建或从一个驱动程序插件 QgfxDriverPlugin 中动态加载。QGfxDriverFactory::keys()返回定义的驱动程序名字符串列表，这个类仅在 Qt/Embedded 中是可用的。

函数 create 为一个匹配参数 key 的驱动程序名创建一个 QScreen 对象，参数 displayId 为显示屏 ID 号。

函数 create（在 src/embedded/qgfxdriverfactory_qws.cpp 中）列出如下：

```
QScreen *QGfxDriverFactory::create( const QString& key, int displayId )
{
    QString driver = key.lower();

#ifndef QT_NO_QWS_VFB
    if ( driver == "qvfb" || driver.isEmpty() )
      return new QVFbScreen( displayId );
#endif
......
//从插件中动态装载
#if !defined(Q_OS_WIN32) || defined(QT_MAKEDLL)
#ifndef QT_NO_COMPONENT
    if ( !instance )
      instance = new QGfxDriverFactoryPrivate;

    QInterfacePtr<QGfxDriverInterface> iface;
```

```
    QGfxDriverFactoryPrivate::manager->queryInterface( driver, &iface );

    if ( iface )
      return iface->create( driver, displayId );
#endif
#endif
    return 0;
}
```

QVoodooScreen 类构造函数（在 src/embedded/qgfxvoodoo_qws.cpp 中）构造一个类对象，列出如下：

```
QVoodooScreen::QVoodooScreen( int display_id )
    : QLinuxFbScreen( display_id )
{
}
```

QLinuxFbScreen 类管理 Linux 帧缓冲(framebuffer)，是 Qt/Embedded 内部使用的类。Linux 加速驱动程序应该从这个类派生，它含有从 Linux 帧缓冲接口读取关于帧缓冲信息的代码，并管理调色板、画面以外的的图形内存和映射帧缓冲接口本身。作为非加速的屏幕光标和 QGfx 的工厂，QLinuxFbScreen 是 QScreen 的派生类，每个 Qt/Embedded 应用程序有一个 QLinuxFbScreen。

QLinuxFbScreen 构造函数（在 src/kernel/qgfxlinuxfb_qws.cpp 中）构造一个 QLinuxFbScreen 对象，参数 display_id 是屏幕 ID 号，列出如下：

```
QLinuxFbScreen::QLinuxFbScreen( int display_id ) : QScreen( display_id )
{
    canaccel=false;
    clearCacheFunc = &clearCache;
}
```

QScreen 类及其派生类管理帧缓冲和调色板，其构造函数（在 src/kernel/qgfxraster_qws.cpp 中）初始化显示的参数，这个构造函数列出如下：

```
QScreen::QScreen( int display_id )
{
    pixeltype=QGfx::NormalPixel;
    data = 0;
    displayId = display_id;
    initted=FALSE;
    entryp=0;
    clearCacheFunc = 0;
    grayscale = FALSE;
}
```

函数 connect 在 src/embedded/qgfxvoodoo_qws.cpp 中）建立具体图形显示设备的帧缓冲，读取图形显示设备的总线配置空间信息，并映射系统内存到指定的虚拟地址，列出如下：

```
bool QVoodooScreen::connect( const QString &spec )
{
    if (!QLinuxFbScreen::connect( spec )) {
      return FALSE;
    }

    canaccel=FALSE;
    /*这是指向 QWS_CARD_SLOT 对应的卡的 256 个字节的 PCI 配置空间信息，从/proc/bus/pci 文件中读出这些信息*/
    const unsigned char* config = qt_probe_bus();

    if(!config)
      return FALSE;
```

```
        unsigned short int * manufacturer=(unsigned short int *)config;
        if(*manufacturer!=0x121a) {
             qDebug("This does not appear to be a 3Dfx card");
             qDebug("Are you sure QWS_CARD_SLOT is pointing to the right entry in "
               "/proc/bus/pci?");
             return FALSE;
        }

        const unsigned char * bar=config+0x10;
        const unsigned long int * addr=(const unsigned long int *)bar;
        //我们期望在配置空间的寄存器的地址指针是一个内存指针，因此做一个简单的检查
        unsigned long int s=*(addr+0); // 第1个寄存器指针
        unsigned long int olds=s;
        if(s & 0x1) {
             return FALSE;
        } else {
             s=s >> 1;
             s=s >> 2;
             s=olds;
             unsigned char * membase;
             int aperturefd;
           /*我们从/dev/mem 映射到寄存器中，/dev/mem 中是以物理内存地址作为视角的（而不是应用程序虚拟地址空间），但包括 PCI 映
射的内存*/
             aperturefd=open("/dev/mem",O_RDWR);
             if(aperturefd==-1) {
     #ifdef DEBUG_INIT
                qDebug("Can't open /dev/mem");
     #endif
                return FALSE;
             }
             s=(s >> 4) << 4;
            /*mmap映射文件 aperturefd 中从偏移 s开始的内容到虚拟内存，返回映射的虚拟内存起始地址。PROT_WRITE 表示内存页可写，
MAP_SHARED 表示与映射这个对象的其他进程共享这个映射，存储到这个区域意味着写入到这个文件*/
             membase=(unsigned char *)mmap(0,0x5fffff,PROT_READ |
                             PROT_WRITE,MAP_SHARED,
                             aperturefd,s);
             if(membase==0 || membase==(unsigned char *)-1) {

                close(aperturefd);
                return FALSE;
             }
            voodoo_regbase=membase;
        }

        qDebug("Detected Voodoo 3") ;

        //我们检测到显卡，这样能安全地使用显卡加速。
        canaccel=TRUE;

        return TRUE;
    }
```

Qt/Embedded 客户端调用函数 connect 来映射帧缓冲，它打开帧缓冲设备、得到设备中屏幕信息、映射帧缓冲到虚拟内存，并且计算屏幕画面的显存尺寸及画面外可用于加速的显存起始地址。

函数 connect 应该被加速的驱动程序重载来映射图形卡的寄存器。为了建立画面外的内存管理，这些驱动程序应该调用这个方法。设备定义在参数 displaySpec 中，如“/dev/fb”。

函数 connect（在 src/embedded/qgfxlinuxfb_qws.cpp 中）列出如下：

```
    bool QLinuxFbScreen::connect( const QString &displaySpec )
    {
        //检查定义的设备
```

```
const int len = 8; // 即“/dev/fbx”的字符长度
int m = displaySpec.find( "/dev/fb" );

QString dev = (m>=0) ? displaySpec.mid( m, len ) : QString("/dev/fb0");
//打开设备
fd=open( dev.latin1(), O_RDWR );
...

fb_fix_screeninfo finfo;
fb_var_screeninfo vinfo;

/*得到固定屏幕信息 */
if (ioctl(fd, FBIOGET_FSCREENINFO, &finfo)) {
     perror("reading /dev/fb0");
     qWarning("Error reading fixed information");
     return FALSE;
}

/* 得到可变的屏幕信息*/
if (ioctl(fd, FBIOGET_VSCREENINFO, &vinfo)) {
     perror("reading /dev/fb0");
     qWarning("Error reading variable information");
     return FALSE;
}

grayscale = vinfo.grayscale;
d=vinfo.bits_per_pixel;
lstep=finfo.line_length;
int xoff = vinfo.xoffset;
int yoff = vinfo.yoffset;
const char* qwssize;
//如果环境变量设置了显示尺寸，读取它，并做限制处理
if( (qwssize=getenv("QWS_SIZE")) && sscanf(qwssize,"%dx%d",&w,&h)==2 ) {
     if ( (uint)w > vinfo.xres ) w = vinfo.xres;
     if ( (uint)h > vinfo.yres ) h = vinfo.yres;
     dw=w;
     dh=h;
     xoff += (vinfo.xres - w)/2;
     yoff += (vinfo.yres - h)/2;
} else {//从设备的可变屏幕信息中得到显示尺寸
     dw=w=vinfo.xres;
     dh=h=vinfo.yres;
}
dataoffset = yoff * lstep + xoff * d / 8;
//qDebug("Using %dx%dx%d screen",w,h,d);

/* 算出屏幕尺寸（单位byte）*/
size = h * lstep;

mapsize=finfo.smem_len;
/*将从 fd 对应文件偏移为 0 的地点开始，映射 mapsize 大小到虚拟内存，即把图形设备的帧缓冲显存映射到虚拟内存*/
data = (unsigned char *)mmap(0, mapsize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
data += dataoffset;

if ((int)data == -1) {
     perror("mapping /dev/fb0");
     qWarning("Error: failed to map framebuffer device to memory.");
     return FALSE;
}

canaccel=useOffscreen();  //空函数

if(mapsize-size<16384) {//显存大小-屏幕尺寸所占显存<16 KB
     canaccel=false; //没有多余显存用于图形加速
```

```
    }

    if(canaccel) {  //可以进行图形加速
        setupOffScreen();  //建立画面之外的内存
    }

    //现在读进调色板
    //如果每个像素点使用8位（对应256色）或4位（对应16色）时，使用调色板
    if((vinfo.bits_per_pixel==8) || (vinfo.bits_per_pixel==4)) {
      screencols= (vinfo.bits_per_pixel==8) ? 256 : 16;
      int loopc;
      startcmap = new fb_cmap;
      startcmap->start=0;
      startcmap->len=screencols;
      startcmap->red=(unsigned short int *)
            malloc(sizeof(unsigned short int)*screencols);
      startcmap->green=(unsigned short int *)
             malloc(sizeof(unsigned short int)*screencols);
      startcmap->blue=(unsigned short int *)
            malloc(sizeof(unsigned short int)*screencols);
      startcmap->transp=(unsigned short int *)
             malloc(sizeof(unsigned short int)*screencols);
      if (ioctl(fd,FBIOGETCMAP,startcmap)) {  //设置调色板
          perror("reading fb cmap");
          qWarning("Error reading palette from framebuffer");
       }
      int bits_used = 0;
      for(loopc=0;loopc<screencols;loopc++) {
         screenclut[loopc]=qRgb(startcmap->red[loopc] >> 8,
                        startcmap->green[loopc] >> 8,
                        startcmap->blue[loopc] >> 8);
         bits_used |= startcmap->red[loopc]
                 | startcmap->green[loopc]
                 | startcmap->blue[loopc];
      }
    //一些帧缓冲驱动程序仅返回8位颜色值，因此，我们不必位移它们
      if ((bits_used & 0x00ff) && !(bits_used & 0xff00)) {
         for(loopc=0;loopc<screencols;loopc++) {
           screenclut[loopc] = qRgb(startcmap->red[loopc],
                          startcmap->green[loopc],
                          startcmap->blue[loopc]);
         }
         qWarning("8 bits cmap returned due to faulty FB driver, colors corrected");
      }
    } else {
         screencols=0;
    }

    initted=true;

    return TRUE;
}
```

上述函数用到了调色板，下面说明调色板。

在计算机的显示系统中，像素的颜色基于 RGB 模型。每一个像素颜色使用 24 位表示，红（B）、绿（G）、蓝（B）三原色分别用 8 位表示，这样 SVGA 适配器可以显示 16.7 M 种颜色。24 位的颜色通常被称为“真彩色”。

真彩色的显示占用大量的显存，例如，一幅 640×480 的真彩色图象需要约 1 MB 的显存。计算机使用调色板来限制颜色的数目，调色板实际上是一个有 256 个表项的 RGB 颜色表，颜色表的每项是一个 24 位的 RGB 颜色值。使用调色板时，在显存中存储的不是 24 位颜色值，而是调色板的 4 位或 8位的索引。这样一来，显示器可同时显示的颜色被限制在 256 色以内，对系统资源的耗费大大

降低了。

显示器可以被设置成 16、256、64 K 或真彩色等显示模式，前两种模式需要调色板。在 16 色或 256 色模式下，程序必须将想要显示的颜色正确地设置到调色板中，这样才能显示预期的颜色。使用调色板的一个好处是不必改变显存中的值，只需改变调色板的颜色项即可快速地改变一幅图像的颜色或灰度。

函数 setupOffScreen 计算画面以外的显存的位置，建立池入口指针表，列出如下：

```
void QLinuxFbScreen::setupOffScreen()
{
    //计算画面以外的显存的位置
    // 建立池入口指针表并且 64bit 对齐
    int psize = size;
    psize += 4096;  //光标数据
    psize += 8;     //为了对齐
    psize &= ~0x7;  //对齐

    unsigned int pos=(unsigned int)data;
    pos += psize;
    entryp = ((int *)pos);
    lowest = ((unsigned int *)pos)+1;
    pos += (sizeof(int))*4;
    entries = (QPoolEntry *)pos;

    //位图可用的画面外显存的开始地址
    cacheStart = psize + sizeof(int)*4 + sizeof(QPoolEntry);
}
```

4.8.6 与图形显示相关的类

与图形显示相关的类如下。

1. QMemoryManager

QMemoryManager 及派生类 QMemoryManagerPixmap 用来管理位图在帧缓存（即显存）中的序号及地址，还可以将未使用的显存存储位图或字体。其成员函数 newPixmap 用来计算新的位图在帧缓存中位图序号及地址，并存入到 QMemoryManagerPixmap 实例中。函数 findPixmap 从 QmemoryManager Pixmap 实例数组中查找到对应 ID 的 QMemoryManagerPixmap 实 例 。

2. QWSRegionManager

窗口由图形区域组成，图形区域由多个矩形组成，类 QWSRegionManager 用于图形区域管理。服务器创建了以./tmp/qtembedded-unknown/QtEmbedded-0 和“r”为 key 的共享内存，并映射到虚拟内存地址 data（它是类 QWSRegionManager 内部使用的地址指针变量），这个共享内存不同于上面以 /tmp/qtembedded-unknown/QtEmbedded-0 和“ m ”为 key 的共享内存。

图形区域在共享内存 data 中存在的格式如下：

| QWSRegionHeader| 第 0个区域 QWSRegionIndex|第 0 个区域的 numRects 个 QRect 对象|.....|第 i 个区域 QWSRegionIndex|第 i 个区域的 numRects 个 QRect 对象|

其中共享内存 data 中的数据格式用到的结构列出如下：

```
class QWSRegionHeader  //区域头
 {
public:
   int maxRegions;  //最大区域数
   int maxRects;  // 最大矩形数
```

```
};

class QWSRegionIndex  //区域序号
{
public:
    int id;
    int revision;  //版本号
    int numRects;  //区域中所含有的矩形数
    int data;
};
```

构造函数 QWSRegionManger 的参数 c=TRUE 时，客户端创建一个 QWSRegionManger 对象得到管理图形区域的共享内存；否则为服务器创建一个 QWSRegionManger 对象，创建共享内存来管理图形区域（一个区域包括多个矩形）。

构造函数 QWSRegionManger （在 src/kernel/qwsregionmanager_qws.cpp 中）列出如下：

```
QWSRegionManager::QWSRegionManager( const QString &filename, bool c) :
    client( c )
{
    data = 0;
    shmId = -1;
    if ( client ) {//客户端得到图形区域的序号
        if ( !attach( filename ) )  //得到共享内存映射的地址data
            qFatal( "Cannot attach region manager" );
        regHdr = (QWSRegionHeader *)(data);
        regIdx = (QWSRegionIndex *)(data + sizeof(QWSRegionHeader));
    } else {//服务器在
        // 服务器需要一个本地的数据拷贝
        regions.setAutoDelete(TRUE);
        regHdr = new QWSRegionHeader;
        regHdr->maxRegions = 0;
        //600*6
        regHdr->maxRects = QT_MAX_REGIONS * QT_RECTS_PER_REGION;
        regIdx = new QWSRegionIndex[ QT_MAX_REGIONS ];
        if ( !attach( filename ) )  //得到映射地址data
        qFatal( "Cannot attach region manager" );
        commit();
    }
}
```

下面对 QWSRegionManger类的用得较多的成员函数进行说明。

对于多进程来说，进程间利用共享内存来通信。这时，函数 attch 根据客户端或服务器来得到或创建共享内存，并映射到进程虚拟地址空间。对于单个进程来说，只需要一个内存空间来存储数据即可。

函数 QWSRegionManager::attach 列出如下：

```
bool QWSRegionManager::attach( const QString &filename )
{
#ifndef QT_NO_QWS_MULTIPROCESS  //多个进程
    //以/tmp/qtembedded-unknown/ QtEmbedded-0 及 r 作为共享内存的key
    key_t key = ftok( filename.latin1(), 'r' );
    if ( !client ) {//服务器
        int dataSize = sizeof(QWSRegionHeader)              // header
            + sizeof(QWSRegionIndex) * QT_MAX_REGIONS // + index
            + sizeof(QRect) * regHdr->maxRects;       // + rects

        shmId = shmget( key, dataSize, IPC_CREAT|0600);  //创建共享内存
        if ( shmId != -1 )
            data = (unsigned char *)shmat( shmId, 0, 0 );  //映射共享内存
    } else {
        shmId = shmget( key, 0, 0 );  //得到共享内存
        // 共享内存中一些数据需要可读写
```

```
        if ( shmId != -1 )
            data = (unsigned char *)shmat( shmId, 0, 0 );  //映射共享内存
    }

    return ( shmId != -1 && (int)data != -1 );
#else  //单个进程
     int dataSize = sizeof(QWSRegionHeader)              // header
             + sizeof(QWSRegionIndex) * QT_MAX_REGIONS // + index
             + sizeof(QRect) * regHdr->maxRects;      // + rects

     data = (unsigned char *)malloc( dataSize );  //分配数据空间
     return TRUE;
#endif
}
```

函数 commit 提交区域表给共享内存，它将各个区域拷贝到共享内存 data 中。一旦所有的区域被修改，应该调用函数 commit 在共享内存中设置新的区域表。区域表并不在每次被修改后拷贝到内存，因为需要考虑在服务器计算新区域时，拷贝区域表会产生大量延迟及不可预测因素。

函数 commit（在 src/kernel/qwsregionmanager _qws.cpp 中）在操作时使用 display 锁锁住了显示用的共享内存，列出如下：

```
void QWSRegionManager::commit()
{
    if ( client ) return;

    QWSDisplay::grab( TRUE ); //锁住 display仅被这个进程访问

    //首先拷贝 rects 区域存在 data 中
    int numRects;
    int offset = 0;
    QRect *r = rects(0); //    得到 data 中偏移位置为 0 的 QRect 对象指针
    for ( int idx = 0; idx < regHdr- >maxRegions; idx++ ) {
         if ( regIdx[idx].id != -1 ) {
            numRects = regIdx[idx].numRects;
            if ( numRects ) {
                 if ( offset + numRects > regHdr->maxRects )  {
                    //在共享内存中区域超界
                    qFatal( "Too many client rects" ) ;
                 }
                 regIdx[idx].data = offset;
              //将 regions 指针数组中的区域对象数据拷贝到 r
                 memcpy( r, regions[idx]->rects ().data()
                                  numRects * sizeof(QRect) ) ;
                 r += numRects;
                 offset += numRects;
            }
         }
    }

    //拷贝区域的头和序号数据到 data 中
    memcpy( data, regHdr, sizeof(QWSRegionHeader) );
    memcpy( data + sizeof(QWSRegionHeader),
         (unsigned char *)regIdx
         regHdr->maxRegions * sizeof(QWSRegionIndex) );

    QWSDisplay::ungrab();//释放 display 的锁
}
```

函数 rects 返回指向 data 中在偏移 offset 位置的 QRects 对象指针，列出如下：

```
QRect *QWSRegionManager::rects( int offset )
{
```

```
    QRect *start = (QRect *)(data + sizeof(QWSRegionHeader) + sizeof(QWSRegionIndex) * QT_MAX_REGIONS);
    return start + offset;
}
```

3. QGfxRasterBase

QGfxRasterBase 类是 QGfxRaster<depth >模板的基类，并含有不依赖于颜色深度的代码。这个类被用于在原始可变颜色深度的帧缓冲上的软件绘制，它可被不同的硬件驱动程序来派生类。为了支持子窗口，它处理区域的剪辑（clip）和可移动的区域。作为是一个 Qt 内部使用的类，仅在 Qt/Embedded 中可用。QWidget 和 QPixmap通过其 graphicsContext()方法返回一个 QGfxRaster 对象指针，这个对象已被初始化了合适的起始点、帧缓冲和 clip 区域。QgfxRasterBase 及其模板子类被作为一个类来考虑，即 QGfxRasterBase 从不被使用，它仅被用来提供一些功能函数。

4.8.7 图形显示

图形显示包括 Qt 的窗口部件 QWidget的显示和位图的显示，实际上 QWidget窗口部件从显示的角度来看也是位图，它们都是通过将图像的像素点所占的字节拷贝到图形设备的帧缓冲区中来实现显示的。为了支持位图的绘制，Qt/Embedded 还提供了 QGfxRaster 类支持在缓冲区中的基本绘制操作。在绘制前，需要建立图形环境，即在指定缓冲区上建立某一颜色深度的 QGfxRaster 类对象。在绘制完成后，将绘制的缓冲区数据拷贝到帧缓冲区上即可显示所绘制的内容。

(1)函数 QWidget::show()

函数 QWidget::show()显示窗口部件及其子窗口部件，如果窗口部件的尺寸或位置已变化， Qt保证在窗口部件显示之前得到 move和 resize 事件。不能重载这个函数，如果必须在窗口部件显示之前改变一些设置，使用 showEvent() 。如果必须使用一些延迟的初始化，使用 polish()。函数 QWidget::show()的调用层次如图 4-20所示。

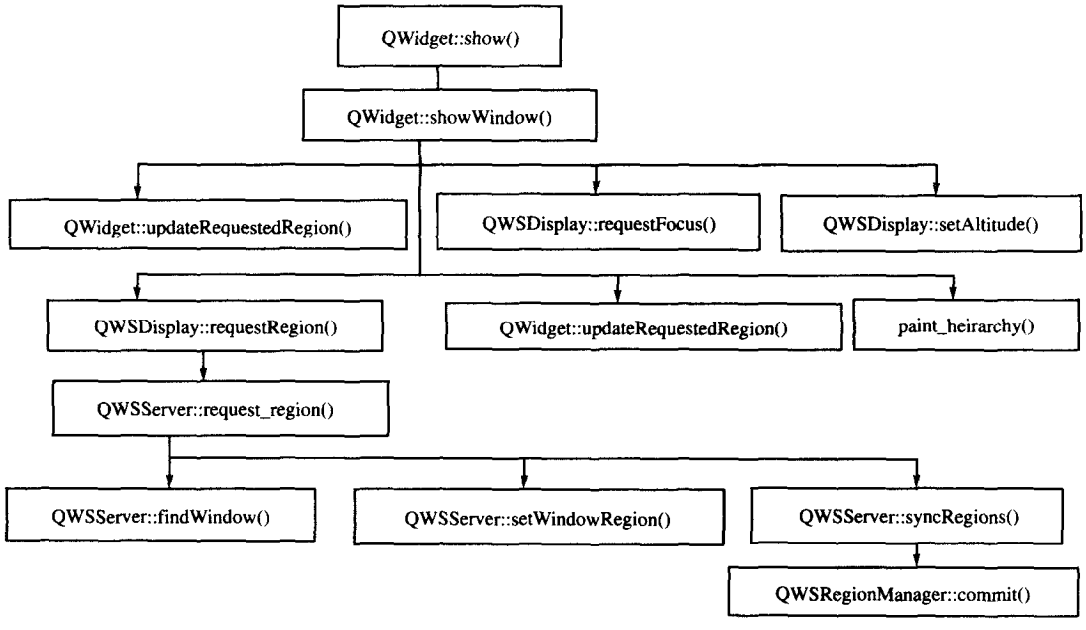


图 4-20 函数 QWidget::show()的调用层次

函数 QWidget::show() 分析如下：

```
void QWidget::show()
```

```
{

    showWindow();

}

void QWidget::showWindow()
{
if ( testWFlags(WType_TopLevel) ) {  // 顶层窗口
//将窗口部件及子窗口部件区域进行变换，使其适合于设备座标空间（如顺时针旋转 90 度）
     updateRequestedRegion( mapToGlobal(QPoint(0,0)) );
     QRegion r( req_region );
#ifndef QT_NO_QWS_MANAGER
    //窗口管理器 qwsManager 提供最大化、最小化，关闭按钮和管理菜单
      if ( extra && extra->topextra && extra->topextra->qwsManager ) {
         QRegion wmr = extra->topextra->qwsManager->region()  ;
         wmr = qt_screen->mapToDevice( wmr, QSize(qt_screen->width()
qt_screen->height()) ) ;
         r += wmr;
      }
#endif
     qwsDisplay()->requestRegion(winId(), r);  //  请求区域显示
     if ( !testWFlags(WStyle_Tool) ) {
        qwsDisplay()->requestFocus(winId(),TRUE);//请求显示焦点
     }
       /*通知服务器设置窗口所在层次（如：顶层），并根据层次升起（ raise)或降低 (lower)窗口 */
     qwsDisplay()->setAltitude( winId()
          testWFlags(WStyle_StaysOnTop) ? 1 : 0, TRUE );

   } else if ( !topLevelWidget()->in_show ) {//  顶层窗口没在显示状态
        updateRequestedRegion( mapToGlobal(QPoint(0,0)) );//更新区域位置
        QWidget *p = parentWidget();
        p->setChildrenAllocatedDirty( geometry(), this );
        p->paintable_region_dirty = TRUE;
        p->overlapping_children = -1;
        paint_heirarchy( this, TRUE );  //绘制窗口部件及其子窗口部件
   }
}
```

函数 requestRegion （在 src/kernel/qapplication_qws.cpp 中）由服务器调用 request_region请求区域显示，列出如下：

```
void QWSDisplay::requestRegion(int winId, QRegion r)
{
   qws_regionRequest = TRUE;
   if ( d->directServerConnection() ) { //   是直接连接到 server
         //直接调用服务器函数请求区域显示
        qwsServer->request_region( winId, r ) ;
   } else {
     //通过发送事件，承诺不在区域外绘制

        QMemArray<QRect> ra = r.rects();  // 得到矩形区域对象，并放入到数组中

        QWSRegionCommand cmd;
        cmd.simpleData.windowid = winId;
        cmd.simpleData.nrectangles = ra.count();
        cmd.setData( (char *)ra.data(), ra.count() * sizeof(QRect), FALSE);
        d->sendCommand( cmd );  // 发送命令给服务器，请求区域显示
   }
   if ( !r.isEmpty() )
       d->waitForRegionAck();//等待确认
}
```

函数 request_region（在 src/kernel/qwindowsystem_qws.cpp 中）是服务器请求区域进行显示的函数，它从窗口映射数组中得到窗口并显示，分析如下：

```
void QWSServer::request_region( int wid, QRegion region )
 {
    QWSClient *serverClient = client[-1];  //得到服务器所在应用程序的客户端
    QWSWindow* changingw = findWindow( wid, 0 );  //  在窗口映射数组中查找到窗口
    if ( !changingw ) {//变化的窗口不存在
      if ( !region.isEmpty() )  // 区域不为空
        //通过队列或 socket 发送事件给用户窗口
         serverClient->sendRegionModifyEvent( wid, QRegion(), TRUE );
      return;
    }
    //变化的窗口存在
    if ( !region.isEmpty() ) //  区域不为空
       changingw->setNeedAck( TRUE );
    bool isShow = !changingw->isVisible() && !region.isEmpty();//  是否在显示
     /*改变窗口 changingw 的请求区域到 region，发送合适的区域变化事件到所有相应的客户端，并等待所有请求得到确认如果参数
changingw 为 0，表示服务器保留的区域发生变化.*/
     setWindowRegion( changingw, region );
     syncRegions( changingw );  //同步窗口进行显示
     if ( isShow )
       emit windowEvent( changingw, Show );
     if ( !region.isEmpty() )
       emit windowEvent( changingw, Geometry );
     else
       emit windowEvent( changingw, Hide );
     if ( focusw == changingw && region.isEmpty() )
       setFocus(changingw,FALSE);
}
```

函数 syncRegions 把需要显示的区域拷贝到共享内存，这个共享内存对应的是帧缓冲区，从而让这个区域显示在屏幕上。该函数分析如下：

```
void QWSServer::syncRegions( QWSWindow *active )
{
    rgnMan->commit();//提交区域表给共享内存，它将各个区域拷贝到共享内存 data 中
    notifyModified( active ); //发送修改事件到 active 窗口及其他窗口
    paintBackground( dirtyBackground );  //绘制背景
    dirtyBackground = QRegion();
}
```

（2）位图显示

类构造函数 QPixmap()（在 src/kernel/qpixmap_qws.cpp 中）初始化位图数据并显示在屏幕上，列出如下：

```
QPixmap::QPixmap( int w, int h, const uchar *bits, bool isXbitmap )
    : QPaintDevice( QInternal::Pixmap )
{                               // for bitmaps only
    //初始化内部数据结构
    init( 0, 0, 0, FALSE, defOptim );
    if ( w <= 0 || h <= 0 )             // create null pixmap
         return;

    data->uninit = FALSE;
    data->w = w;
    data->h = h;
    data->d = 1;
    data->rw = qt_screen->mapToDevice( QSize(w,h) ).width();
    data->rh = qt_screen->mapToDevice( QSize(w,h) ).height();
    data->hasAlpha = FALSE;
```

```
    uchar *flipped_bits;
    if ( isXbitmap ) {
         flipped_bits = 0;
    } else {                          // not X bitmap -> flip bits
         flipped_bits = flip_bits( bits, ((w+7)/8)*h );
         bits = flipped_bits;
    }

    if ( qt_screen->isTransformed() ) {  //屏幕显示变换，如旋转90度
         int bpl = isXbitmap ? (w+7)/8 : ((w+31)/32)*4;
         QImage img( (uchar *)bits, w, h, 1, bpl, 0, 0, QImage::LittleEndian );
         convertFromImage( img, MonoOnly );
         if ( flipped_bits )
            delete [] flipped_bits;
         return;
    }

    //找到一块匹配显示区域大小的显存块
    data->id=memorymanager->newPixmap(data->rw,data->rh,data->d, optimization());
    if ( data->id == 0 ) {
         //超出内存，创建空位图
         data->w = data->h = 0;
         return;
    }
    uchar *dest;
    int xoffset,linestep;
    //在hash表pixmap_map中找到指定id的显存位图，位图在显存中的地址为dest
    memorymanager->findPixmap(data->id,data->rw,data->d,&dest,&xoffset,&linestep);

    Q_ASSERT((xoffset&7) == 0); // if not, we need to fix this to do a bitblt
    dest += xoffset/8;

    //将用户的位图数据bits拷贝到显存位图，即可显示
    uchar *src = (uchar*)bits;
    for (int row = 0; row < h; row++)
    {
         memcpy(dest, src, (w+7)/8);
         dest += linestep;
         src += (w+7)/8;
    }

    if ( flipped_bits )                  // Avoid purify complaint
         delete [] flipped_bits;
}
```

函数初始化内部的数据结构 data，从显存中找到一块匹配的显存块，分析如下：

```
void QPixmap::init( int w, int h, int d, bool bitmap, Optimization optim )
{
    static int serial = 0;
    int dd = defaultDepth();

    if ( !qws_pixmapData )
       qws_pixmapData = new QList<QShared>;

    if ( optim == DefaultOptim )         // use default optimization
      optim = defOptim;

    /*创建并初始化QPixmapData*/
    data = new QPixmapData;
    Q_CHECK_PTR( data );

    if ( qws_trackPixmapData )
         qws_pixmapData->append( data );
```

```
    memset( data, 0, sizeof(QPixmapData) );
    data->id=0;
    data->count  = 1;
    data->uninit = TRUE;
    data->bitmap = bitmap;
    ......

    data->rw = qt_screen->mapToDevice( QSize(w,h) ).width();
    data->rh = qt_screen->mapToDevice( QSize(w,h) ).height();

    //在显存中找到显示区域大小的显存块
    data->id=memorymanager->newPixmap(data->rw, data->rh, data->d, optim);
    if ( data->id == 0 )
         data->w = data->h = 0; // 超出内存，创建一个空位图
}
```

函数 newPixmap 从显存中找到指定显示区域大小的显存块，若找不到，则从内存中分配一块，并把分配的显存块编上序号放入 hash 表。该函数分析如下：

```
QMemoryManager::PixmapID QMemoryManager::newPixmap(int w, int h, int d
                                int optim )
{
    uchar* data;
    PixmapID id;

    int xoffset;

    const int test_offset=0;
    //计算宽 w、色 深 d、高 h显示区域所占的大小 (byte)
    int siz = calcLineStep( w, d, TRUE ) * h;

    //在显存中查找这个显示区域大小的显存块，并返回起始地址
    data=qt_screen->cache(siz, optim);

    if ( data ) {//找到了需要的显存块
         xoffset = 0; // XXX
         // 使用下一个 next_pixmap_id 作为 id 来标识这个显存块
         id = ++next_pixmap_id;
         next_pixmap_id++; /  / stay even
         //    qDebug( "pixmap in VRAM %d"     id );
    } else {   //  如果在显存中未找到，则从主存中分配
         //在主存中可能有不同的对齐方式
         siz = calcLineStep( w, d, FALSE ) * h;
         xoffset = test_offset; // for testing
         data = (uchar*)malloc(siz);  //  分配内存
         if ( !data )
            return 0; //out of memory
            // even id
         id = next_pixmap_id;  //  得到 id，它是奇数的 id
         next_pixmap_id += 2; //保持奇数
    }

    memset((void *)data,0,siz); //  清空显存块或内存块

    //初始化 mmp，它存有分配的显（内）存块的起始地址及偏移，并放入一个 hash 表中
    QMemoryManagerPixmap mmp;
    mmp.data = data;
    mmp.xoffset = xoffset;
    pixmap_map[id]=mmp;

    return id;
}
```

（3）图形环境

图形环境是指在一个指定的缓存区建立某一颜色深度的 QGfxRaster 对象，绘制操作经 QGfxRaster 对象的基本绘图成员函数变成缓存区中的字节。在需要显示时，把这一缓存区内容拷贝到帧缓冲区中即可显示。

下面以屏幕抓图操作函数 grabWindow 为例说明图形环境在绘制中的应用。

函数 grabWindow 查找到窗口 ID 对应的 QWidget，建立图形环境。并将显存中的图像拷贝到位图中返回，从而达到“抓屏”的作用。该函数列出如下：

```
QPixmap Qpixmap::grabWindow( Wid window, int x, int y, int w, int h )
{
   Qpixmap pm;
   Qwidget *widget = Qwidget::find( window );
   if ( widget ) {
      if ( w <= 0 || h <= 0 ) {
         if ( w == 0 || h == 0 )
            return pm;
         if ( w < 0 )
            w = widget->width() - x;
         if ( h < 0 )
            h = widget->height() - y;
      }
      pm.resize(w, h);
      QGfx *gfx=pm.graphicsContext(); //建立图形环境
      if ( gfx ) {
           //为 blt 操作设置 alpha 通道类型：忽略 alpha 混合
        gfx->setAlphaType(QGfx::IgnoreAlpha);
       gfx->setSource(widget);  //  设置图像的宽、高等参数
       //将 (x, y) 和 (x+w, y+h) 座标内区域拷贝到图形环境的缓冲区中 (0,0) 坐标处
        gfx->blt(0,0,w,h,x,y);
        delete gfx;
      }
   }
   return pm;
}
```

函数 graphicsContext 创建位图环境，它在一个指定的缓存中建立 QGfxRaster 对象。这样，将绘制的图形经 QGfxRaster 对象的基本绘制函数变成这个缓存中对应的字节。函数 graphicsContext 是类 QPixmap 内部使用的函数，列出如下：

```
QGfx * QPixmap::graphicsContext(bool) const
{

   uchar * mydata;
   int xoffset,linestep;
   //找到 data 结构对应位图在显存中的地址、偏移和每行像素点所占显存大小 linestep
   memorymanager->findPixmap(data->id,data->rw,data->d,&mydata,&xoffset,&linestep);

   /*以 mydata 上根据颜色深度 depth()创建对应 的 QGfxRaster 对象，这样可以使用 QGfxRaster  的成员函数在 mydata 缓存中绘制基本图形*/
   QGfx * ret=QGfx::createGfx( depth(), mydata, data->w,data->h, linestep );
   if(data->d<=8) {// 如果颜色深度小于 8 位
        if(data->d==1 && !(data->clut)) {
           data->clut=new QRgb[2];
           data->clut[0]=qRgb(255,255,255);
           data->clut[1]=qRgb(0,0,0);
           data->numcols = 2;
        }
        if ( data->numcols )//设置颜色映射
```

```
            ret->setClut(data->clut,data->numcols);
        }
        return ret;
    }
```

4.9 运行 Qt/Embedded 应用程序

可以在配置 Linux 帧缓冲的 PC 机或嵌入设备上运行 Qt/Embedded 应用程序。默认情况下没有配置帧缓冲驱动，因此需要加入帧缓冲驱动程序并重编译内核，帧缓冲驱动程序才能正常工作。 PC 机上配置了在 PC 机上帧缓冲驱动程序时，启动 PC 机时会在屏幕上出现一个小企鹅图标。

在 PC 机上，如果你没有一个配置可工作的 Linux 帧缓冲，则可以使用 Qt/Embedded 虚拟帧缓冲 qvfb，或者运行 Qt/Embedded VNC 服务器。

一个 Qt/Embedded 应用程序需要运行一个主应用程序或其自己作为主应用程序运行，主应用程序包含服务器，这个服务器主要负责管理顶层窗口区域、指针设备和键盘的输入。

如果需要使应用程序可以成为主应用程序运行，可以设置 QApplication::GuiSever 类型或以-qws 命令行参数运行应用程序来构造 QApplication 对象。

假定已正确配置了 Linux 帧缓冲并且未设置运行主应用程序，切换到控制台并选择一个样例应用程序运行，如 example/widgets。确信$QTDIR 设置到安装 Qt/Embedded 的目录，并且添加$QTDIR/lib 目录到$LD_LIBRARY_PATH 上。环境变量设置如下：

```
export QTDIR=$HOME/qt-version
export LD_LIBRARY_PATH=$QTDIR/lib:$ LD_LIBRARY_PATH
```

使用-qws 选项运行应用程序，方法如下：

```
cd $QTDIR/examples/widgets
./widgets -qws
```

Qt/Embedded 可以通过运行多个主应用程序进程，并允许多个显示同时进行，这通过使用-display 命令选项或设置$QWS_DILPLAY 环境变量得到。-display 参数的语法如下：

```
[gfx driver][:driver specific options][:display number]
```

例如，如果要在 fb1 上使用 mach64 驱动程序，作为 display2 来显示，则命令如下：

```
$ ./launcher -display Mach64: /dev/fb1:2
```

还可使用下面一些命令执行其他测试。

（1）切换到控制台 1 并运行 launcher，命令如下：

```
$ cd examples/launcher
$ ./launcher
```

（2）切换到控制台 2 并运行另外一个 launcher，命令如下：

```
$ cd examples/launcher
$ ./launcher -display:1
```

（3）按 Ctrl+Alt+F1 键回到 display 0。

仅主应用程序进程需要明确定义 driver/device 部分，Client 在连接时从主应用程序得到所需要的信息。因此，一旦有一个使用特定驱动程序的主服务器在运行，则仅能使用 “client -display:n” 来使用

display n。

使用下面的环境变量来设置使用的指针设备（如鼠标和触摸屏等）：

```
export QWS_MOUSE_PROTO=IntelliMouse
```

4.10 字符输入

在 Qt/Embedded 的客户/服务器内部，键盘每个键的按下和释放作为一个 QWSKeyEvent 事件被发送。一个 QWSKeyEvent 事件包含下面的成员。

（1）Unicode：Unicode 值。

（2）Keycode：在 qnamespace.h 中有定义的 Qt keycode 值。

（3）Modifier：由 bit位组成 Qt::ShiftButton、Qt::ControlButton 和 Qt::AltButton 域。

（4）is_press：如果键被按下，为 TRUE；如果键被释放，为 FALSE。

（5）is_auto_repeat：如果这个事件被自动重复引起，为 TRUE。

当 Server接收到一个 key 事件，这个事件被发送给每个客户进程，客户进程负责处理 key 事件，并且把该事件发送到合适的窗口。key 事件可能来自几个不同的源，分别说明如下。

（1）键盘驱动程序

一个键盘驱动程序从一个设备读取数据，并把 key 事件给 Server。键盘驱动程序当前被编译进库中，它们被定义在文件 src/kernel/qkeyboard_qws.cpp 中。当前支持的驱动程序如下。

- QWStyKeyboardHandler：表示从系统控制台（tty）进行键盘输入。
- QWSVr41xxButtonsHandler：表示从 Cassiopeia 型号 PDA 的按钮进行键盘输入。
- QWSVFbKeyboardHandler：表示从虚拟帧缓冲进行键盘输入。

键盘驱动程序都遵循同样的模式，它们从设备读取键盘数据，找出哪个键被按下，然后用按键信息调用静态函数 QWSWerver::processsKeyEvent()。

当前，控制台键盘驱动程序也处理控制台切换（Ctrl+Alt...F10）和关闭控制台（Ctrl+Alt+Backspace）

为了给一个新设备添加键盘驱动程序，需要创建一个 QWSKeyboardHandler 的派生类，并在 qkeyboard_qws.cpp 文件的 QWSServer::newKeyboardHandler() 函数中实例化这个派生类。

（2）键盘事件过滤器（输入方法）

当 Server 从一个键盘驱动程序中接收一个 key 事件，它首先把它传送给一个过滤器。这个过滤器能被用来运用输入方法，提供键盘上没有的字符的输入。为了创建一个输入方法，从 QWSServer::KeyboardFilter（在 src/kernel/qwindowsystem_qws.h 中）派生子类，并实现虚拟函数 filter()。如果 filter() 返回 FALSE，key 事件将被送到 Client（使用 QWSServer::sendKeyEvent()）；如果 filter()返回 TRUE，key 事件将被停止。为了产生新的 key 事件，使用 QWSServer::sendKeyEvent()。不要使用 processKeyEvent()，因为这将导致无限制的递归。

安装一个键盘过滤事件过滤器，使用 QWSServer::setKeyboardFilter()，当前一次仅能安装一个过滤器。过滤操作必须在 server 进程中被处理。

（3）笔输入

key 事件不一定来自一个键盘设备，Qt/Embedded 的服务器进程可以在任何时候调用 QWSServer::sendKeyEvent()。典型处理是弹出一个 widget 窗口，用户使用指针设备来选择字符。

在 Qtopia 中包含如下 3 个输入窗口部件的例子。

- 一个简单的手写识别样例。
- 一个虚拟键盘样例。

- 一个 Unicode 输入的 widget 窗口。

4.11 增加一个加速显示卡驱动程序到 Qt/Embedded

Qt/Embedded 有能力使用硬件加速，为了增加一个 PCI 或 AGP 驱动程序，必须执行下面的步骤。

（1）定义一个加速的 QLinuxFbScreen 的派生类，为此使用 QVoodooScreen::connect 映射其寄存器。使用 qt_probe_bus 得到 PCI 配置空间的指针，应该检查正在指向一个正确的设备（使用 PCI 设备/生产商 ID 信息来检查）。

（2）使用 PCI 配置空间在物理内存上定位设备加速器寄存器，并且从/dev/memmmap 映射合适的区域。没有必要映射 framebuffer，QLinuxFbScreen 将做映射。如果在任何点发生错误，将返回 FALSE。

QVoodooScreen::initDevice 将仅被 QWS server 调用，并且保证在任何绘制操作之前调用。这是一个设置寄存器和状态的好地方，每个正在连接的客户端将调用 connect。

（3）从 QGfxRaster 定义一个派生类，这是实际的绘制代码处，未被硬件实现的任何功能被传递到 QgfxRaster 中用软件来实现。使用 optype 变量确信加速和未加速的操作是同步的（如果你通过软件画一个区域，在这个区域中硬件加速器还在绘画，那么你的绘画操作将出现错误的次序）。 optype 将保存在可共享内存中，并且被未加速的操作设置为 0，加速操作应该将其设置为 1。当一个软件图形操作被请求时，optype 设置为 1。QGfxRaster::sync() 被调用，你应该提供自己的应用来等待图形引擎空闲。lastop 也是可用的，并被在共享空间。它将不被仅有软件的 QGfx 设置，并可用来存储你的上一次操作的类型（如绘制一个矩形）。当很多同样的操作系列执行时，可以减少下一次操作的建立部分。

在任何寄存器、 lastop 或 optype 被访问并且在 ungrabbed()函数之前，所有的绘制操作应该通过一个 QWSDisplay::grab()被保护。以阻止两个应用程序同时尝试访问加速器，从而可能导致机器的死锁。这种情况是可能的，源数据不在显卡上。应该检查，并且如果有必要，应阻止软件的访问。

注意 QGfxRaster 支持的一些特征并不直接被 QPainter 支持，如在 32 位数据的 alpha 通道和 stretchBlt。这些特征被 Qt 使用，stretchBlt 加速 QPixmap::xForm 和 drawPixmap 到一个变形的 QPainter，alpha 通道加速支持 32 位位图。

（4）如果你愿意，定义一个加速的 QScreenCursor 的派生类，restoreUnder、saveUnder、drawCursor 和 draw 应该定义为空操作。利用 set、move、show 和 hide，在 framebuffer 可见部分的末尾将剩下 4 KB 给 cursor（鼠标箭头），如（宽×高×深）/8。

（5）在 QScreen 的派生类中使用 initCursor 和 createGfx，利用 useOffscreen，并且如果可使用 offscreen 显卡内存，返回 true。

（6）使用函数 qt_get_screen_mychip，它简单地返回一个新的 QMychipScreen。

（7）在 qgfxraster_qws.cpp 文件中添加驱动程序到 DriverTable 表中，如

```
{"MyChip", qt_get_screen_mychip, 1}
```

第 1 个参数是 QWS_DISPLAY 使用的名字，用来申请加速驱动程序。

（8）运行新的驱动程序，输出环境变量如下：

```
export QWS_DISPLAY = MyChip
```

例如，可将 MyChip 设为/dev/fb，表示申请了一个不同于/dev/fb0 的 Linux 帧缓冲运行程序。

如果驱动程序不是 PCI 或 AGP 接口，则需要从 QScreen 派生类，而不是 QlinuxFbScreen。并且使用类似于 QLinuxFbScreen 的函数，处理也类似，最完整的驱动程序例子是 qgfxmach64_qws.cpp。*qgfxvoodoo_qws.cpp* 可能提供了一个较小，但更容易理解的驱动程序。

4.12 Qt/Embedded 作为 VNC 服务器

VNC 协议允许从网络上的任何地方查看并与计算机的显示屏交互。为了在 Qt/Embedded 中使用这种方法，使用 -vnc 选项配置 Qt，并确信支持 16bit 的显示。运行应用程序如下：

```
application -display VNC:0
```

这样，运行的 VNC 客户端便指向这台运行应用程序的机器。例如，使用 X11 VNC 客户端在同一台机器上查看应用程序，命令如下：

```
vncviewer localhost:0
```

默认情况下，Qt/Embedded 将创建 640×480 分辨率的显示。可通过 QWS_SIZE 环境变量改变这个设置，如 QWS_SIZE = 240×320。

VNC 客户端在许多操作系统中可用，如 X11、Windows、Amiga、DOS 和 VM 等。

第 5 章　Qtopia 的核心技术

Qtopia 是嵌入 linux 的桌面系统，它建立在 Qt/Embedded 之上。Qtopia 分为开放源代码版本与收费版本，其中开放源代码版本提供了 PDA 的桌面系统基本源代码；收费版本还包括了手机模块代码部分。

本章介绍 Qtopia 开放源代码版本 Qtopia 核心技术及类库，着重分析插件的装载过程、应用程序快速启动器的机制、类 MimeType 及编写输入法的方法等。

5.1　Qtopia 简介

Qtopia 是 Trolltech 公司为采用嵌入式 Linux 操作系统的电子设备开发的应用平台，Trolltech 公司提供手机版 Qtopia 和 PDA 版 Qtopia 消费电子产品平台。Qtopia 手机版和 Qtopia PDA 版是针对移动电话和 PDA 的解决方案，而 Qtopia 消费电子产品平台（Qtopia CEP）则是一套高层次开发平台，使得制造商能够在手写笔和键盘驱动的设备上创建自定义的环境，包括电视机 Web Pad 无线联网机顶盒等。

Qtopia 平台包含在 Qt/Embedded 基础上构建消费电子产品所需的所有功能，包括应用程序管理器、快速应用程序启动、应用程序安装、红外线/蓝牙通信、本地化框架、基于通用插件的框架、警报服务器，以及 PIM 数据 API 等功能。

Qtopia 是嵌入式 Linux 设备的平台，它还提供给终端用户一套核心应用程序，如 PIM（个人信息管理）、地址本、日历和媒体播放器等应用程序。

Qtopia 提供给开发者一个面向对象的 API 来给嵌入设备编写应用程序，这些 API 与 Qt/X11 下是一致的，因而你的 Qt 应用程序可以在 Qtopia 平台上运行。

Qtopia 平台包括运行的嵌入设备和 PC 机上的软件，运行在 PC 机上软件称为“Qtopia Desktop”，允许用户在嵌入设备和 PC 机 Qtopia Desktop 间同步数据。

Qtopia 软件分为 4 个主要层次，如图 5-1 所示。

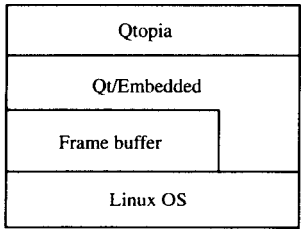


图 5-1　Qtopia 软件的 4 个层次

Java 程序在 Qt/Embedded 层上面，其他程序（如 Opera 网页浏览器和 Handcom 办公软件）建立在 Qt/Embedded 和 Qtopia 上面。

5.1.1　Qtopia PDA

Qtopia PDA 版是一个强大的平台，专用于基于 Linux 操作系统的 PDA。许多 PDA 设备都已采用

了 Qtopia。Qtopia PDA 版具有可定制的用户界面，支持更大的屏幕尺寸，以及横向和纵向布局。

Qtopia 的组件包括窗口操作界面、游戏和多媒体、同步框架、 PIM 应用程序、Internet 应用程序、开发环境、输入法、Java 集成、本地化支持、个性化选项和无线支持等。

5.1.2 Qtopia 手机版

Qtopia 手机版是 Trolltech 公司专门针对移动电话市场推出的 Qtopia 手机版和嵌入式 Linux 的组合，它提供所有基本的 Qtopia 应用程序以及专用于手机的用户界面。 Qtopia 手机版的架构包含 4 大组件，即平台、应用程序、开发环境及同步框架。此外， Qtopia 手机版还支持两种类型的用户交互方式，即键盘和触控屏。

Qtopia 手机版的内容如下。

（1）手机用户界面

Qtopia 手机版的用户界面面向移动电话用户设计，屏幕最终用户可以对其进行个性化设置，制造商和网络运营商也可对其进行自定义。

（2）应用程序

Qtopia 手机版包含许多手机应用程序，如 PIM 工作辅助、电子邮件、游戏、支持 MP3 和 MPEG 等的媒体播放器，以及众多用于手机个性化的实用工具。

（3）平台

Qtopia 手机版在占用较少内存的情况下实现智能手机的功能。

（4）同步

用户可以将手机与 Qtopia Desktop 的一个跨平台桌面 PIM 套件或 Microsoft® Outlook 同步。Qtopia 手机版可以同步的内容包括通信录、日历事件、待办事宜，以及多媒体文件和文档。

（5）开发环境

Trolltech 为开发人员提供了一套工具，使其能够自定义和扩展功能。 Qtopia 手机版的开发环境允许开发人员能够轻松地在桌面系统上进行开发，而后针对目标设备进行交叉编译。

手机版 Qtopia 由 6 个部件组成，即插件（plugin）、服务器（Server）、客户端（Client）和内建的应用程序、 file、modem，以及帧缓冲，其关系图如图 5-2 所示。

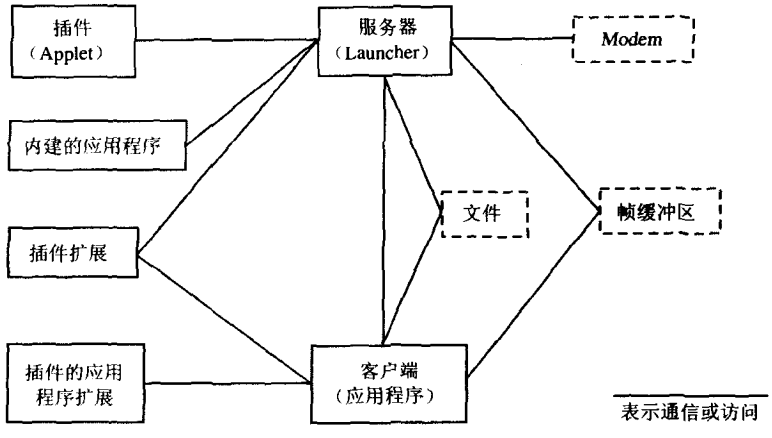


图 5-2 Qtopia 手机版部件间的关系

Qtopia 的 launcher/server 是用来控制窗口系统、进程间通信、开始所有应用程序和其他核心任务。

插件管理器确保在应用程序/服务器启动时装载不同插件类型，插件为应用程序增加了新的功能或提供开发新特征的通道。在图中的文件包括文档、配置和系统文件，以及应用程序特定的文件。手机

版 Qtopia提供了标准的 GSM AT 命令并支持专有命令，用于支持 GPRS 和外部的串行连接器。

5.1.3 手机库

Qtopia 中的手机版本不是 GPL 授权，因此在基于 GPL 授权的 Qtopia 中没有手机库。手机库用来访问硬件、实现打电话和接听电话、收发 SMS 消息，以及访问 SIM 卡等。基本手机库的系统框架如图 5-3 所示。

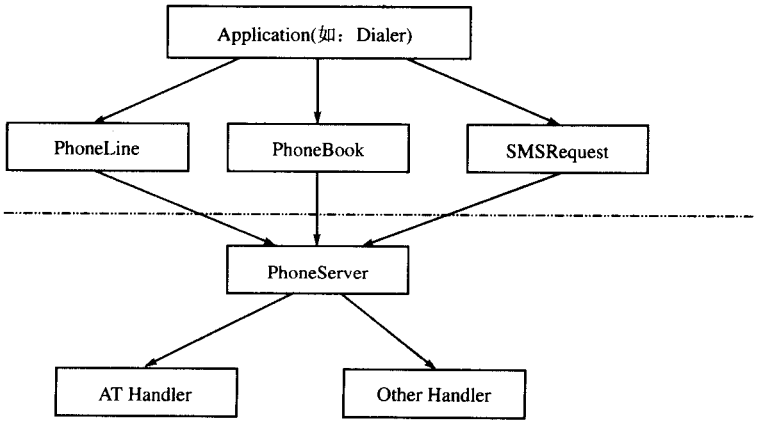


图 5-3 基本手机库的系统框架

诸如拨号器的应用程序使用 C++的 API 来访问电话线模块，以及 SIM 卡上的电话本或 SMS 消息系统。

来自应用程序的请求被传到电话服务器，电话服务器再把请求传给电话功能模块。这样，允许 SMS 消息客户端和拨号器同时访问电话硬件。

电话服务器接着传递请求设备处理器，它与硬件设备通信并影响到请求，响应和状态消息沿相反的方向传回。

当前，我们应用基于 AT 命令集的设备处理器，实际的设备通常通过串口访问。

下面以打电话和接收 SMS 消息两个例子来说明请求和响应的流程。

1. 拨一个号码

为拨一个号码，执行如下流程。

（1）创建一个 PhoneLine 的实例。

（2）调用 PhoneLine::createCall 方法得到代表打电话的 PhoneCall 类的一个实例。

（3）调用 PhoneCall::dial 方法拨一个号码，PhoneCall::dial 发送请求到电话服务器。

正常情况下，每个应用程序仅有一个 PhoneLine 的实例。如果有呼叫被 hold 或有一个多方电话会议在进行，将可能有多个 PhoneCall 的实例。

phone 库使用 QCop 传递拨号请求到电话服务器，电话服务器将执行下面的操作。

（1）获得代表 AT 命令处理器（控制电话线）的 PhoneLineAt 实例。

（2）调用 PhoneLineAt::createCall 方法来创建一个 PhoneCallAt类实例。

（3）调用 PhoneCallAt::dial 方法拨出号码。

（4）PhoneCallAt::dial 发送 ATD 命令给电话设备来使拨号请求生效。

2. 接收 SMS 消息

当 AT 命令处理器启动时，它使用 AT+CNMI 命令注册一个新的消息指示器。当一个新的消息到

达设备时，将引起电话设备发送+CMTI指示。

在代码中，+CMTI 将被 SMSRequestAt::notification 检测，并且引起 AT 命令处理器初始化一个消息计数检查（SMSRequestAt::check）。一旦检查完成，消息的数量通过 SMSRequestPrivate::messageCount 信号被传递到应用程序，这个信号通过 QPE/Phone QCop 信道被广播。应用程序在这个信道上能侦听接收到新消息的通知（notification），通常邮件客户端应该随后取回消息内容。

当邮件客户端收到这个信号时，它调用 SMSRequest::firstMessage 和 SMSRequest::nextMessage 方法从收到的队列中取回 SMS 消息内容，每个消息通过 SMSRequestPrivate::fetched 信号被返回到应用程序。

最后，一旦应用程序收到消息，它将使用 SMSRequest::deleteMessage 从进来的队列中删除这个消息。

我们总是使用信号来传回消息和状态报告给应用程序而不是函数的返回，这是因为整个电话系统是异步的。当请求产生时，信息可能是不可用的，因此必须以后传递信息。

设备支持的 AT 控制命令必须遵循 GSM07.07 规范。

5.1.4 Qtopia Desktop

Qtopia Desktop 用来存储用户数据、加载应用程序并在 PC机与 Qtopia 设备之间同步传输数据，它是一个 PIM 中心。Qtopia Desktop 有支持 Windows、MacOS 和 Linux 操作系统的版本。

PIM 是个人信息管理（Personal Information Managenment）的缩写，Qtopia手机版中的 PIM 功能通过下面的应用程序实现。

（1）主页屏幕

显示未接电话语音邮件文本信息和电子邮件。

（2）通信录

- 输入和同步通信录。
- 记录电话号码电子邮件地址等。
- 输入照片或图像用于来电显示。

（3）日历

- 输入和查看会议约会和生日信息。
- 包括日期、星期和月份视图。

（4）待办事宜

- 创建有待完成的任务列表。
- 包括设定优先级和选择到期日。

Qtopia Desktop 的 PIM功能允许用户在 PC 机上创建和编辑通信录、约会和待办事宜等，用户可在 Qtopia Desktop和 Qtopia 手机版之间轻松地同步 PIM 数据。

Qtopia Desktop 可连接到 Qtopia 应用程序所在的网站，检索可下载的应用程序，并列出已安装和未安装的 Qtopia 应用程序。

5.2 信道和消息

系统信道用来广播消息，如 shutdown()、quit()及 close()等。每个应用程序通过系统信道接收到广播消息，系统信道的连接槽函数 systemMessage 用来处理广播消息。它根据消息分类进行处理，如果是系统管理信息，将直接处理，如 quit()将引起应用程序退出；如果是窗口处理消息，将转发给相应的窗口，如 setVolume(int,int)消息将被转发。

应用程序名信道用来传送应用程序特定的信息，如 raise()和 flush()等。应用程序接收自己应用程

序名信道中的消息，对应槽函数 pidMessage()来处理或转发这些消息。

信道允许应用程序彼此间通信，消息通过使用 QCopEnvelope发送，应用程序连接到 QCopChannel 来接收消息。在 Qtopia 中，信道名以“ QPE/ ”开始，消息名是一个函数名加上函数参数类型，不能有空格。在程序代码中发送消息的方法如下：

```
QCopEnvelope e(channelname, "messageName(Type1,Type2,Type2)");
e << parameter1 << parameter2 << parameter3;
```

QCop 消息系统使用了事件处理循环，因此如果创建了一个 QCop 消息，则其将直到下一个 qApp::processEvents 时才会被发送。

系统范围的信道有 QPE/System 系统信道、QPE/TaskBar 任务栏信道、QPE/Card 插卡信道、QPE/Network 网络信道、Qt/Tray 系统托盘信道及应用程序信道，下面分别说明各个系统统信道中的消息。

（1）QPE/System 系统信道

QPE/System 系统信道的消息说明如表 5-1 所示。

表 5-1 QPE/System 系统信道的消息

消息名	说 明
busy()	引起一个忙等待指示器显示给用户
notBusy()	取消忙等待指示器
linkChanged(QString)	当链接变化时发送，由 FileManager::saveFile()和其他的函数发送，并被 Launcher（应用程序启动器）接收
applyStyle()	当用户改变风格时发送，由 QPEApplication 接收，并且应用程序按照新的风格重画
quit()	关闭所有的应用程序
shutdown()	显示关闭的窗口
restart()	重启动 Qtopia 环境
execute(QString)	运行或升起给定的应用程序
execute(QString,QString)	运行或升起给定的应用程序，并传递一个参数。注意应考虑使用 QPE/Application/appname 信道
setMouseProto(QString)	动态改变鼠标协议$QWS_MOUSE_PROTO
setKeyboard(QString)	动态改变键盘$QWS_KEYBOARD
addAlarm(QDateTime,QCString,QCString,int)	加闹钟，内部使用
deleteAlarm(QDateTime,QCString,QCString,int)	删除闹钟，内部使用
setDefaultRotation(int)	设置默认旋转度数，内部使用
grabKeyboard(QString)	捕获键盘，内部使用
timeChange(QString)	时间改变，内部使用
clockChange(bool)	时钟改变，内部使用
setScreenSaverInterval(int)	设置屏幕保护时间间隔，内部使用
language(QString)	设置语言，内部使用
resetPowerTimeout(int)	重启动定时，内部使用

（2）QPE/TaskBar 任务栏信道消息

QPE/TaskBar 任务栏信道的消息如表 5-2 所示。

表 5-2 QPE/TaskBar 任务栏信道的消息

消息名	说 明
message(QString)	显示状态信息给用户，这些信息通常在任务栏上出现短时间后消失，内部使用
showInputMethod()	显示输入方法，内部使用
hideInputMethod()	隐藏输入法，内部使用
reloadInputMethods()	重装载输入法，内部使用
reloadApplets()	重装载 java 程序，内部使用

（3）QPE/Card 插卡信道消息

QPE/Card 插卡信道的消息如表 5-3 所示。

表 5-3 QPE/Card 插卡信道的消息

消息名	说 明
mtabChanged()	当/etc/mtab 发生变化时发送
stabChanged()	当/var/run/stab，/var/state/pcmcia/stab，或/var/lib/pcmcia/stab 发生变化时发送

（4）QPE/Network 网络信道消息

QPE/Network 网络信道的消息如表 5-4 所示。

表 5-4 QPE/Network 网络信道的消息

消 息 名	说 明
up()	当一个网络连接正运行时发送的消息
down()	当一个正运行的网络连接关闭时发送的消息
failed()	当一个网络连接启动失败时发送的消息
available(QStringList)	当可用的网络连接变化时或作为对 announceChoices()消息的响应而发生的消息
announceChoices()	强制网络监视器发送可用的连接表

（5）Qt/Tray 系统托盘信道消息

这个信道允许 Qt 应用程序（不仅仅是 Qtopia 应用程序）以交互平台方式显示系统托盘状态，但它目前不被支持，TaskbarAppletInterface 是系统托盘应用程序更有力的替代。

Qt/Tray 系统托盘信道的消息如表 5-5 所示。

表 5-5 Qt/Tray 系统托盘信道的消息

消息名	说 明
popup(int,QPoint)	托盘弹起时发送的消息
doubleClicked(int,QPoint)	双击托盘发送的消息
clicked(int,QPoint)	点击托盘发送的消息
remove(int)	删除托盘发送的消息
setIcon(int,QPixmap)	在托盘上选择图标发送的消息
setToolTip(int,QString)	在托盘上显示工具提示消息框时发送的消息

（6）应用程序信道

每个应用程序在名为 QPE/Application/appname 的信道上监听，appname 是应用程序的执行名字。这个信道上的标准消息如表 5-6 所示。

表 5-6 应用程序信道的消息

消息名	说 明
setDocument(QString)	引起主文档部件的槽函数 setDocument(const QString&)被调用，这是个内部使用的消息
nextView()	连接到硬件按钮的应用程序如果是可见的，则收到这个消息
raise()	这个消息引起主文档部件和当前的模式窗口部件被升起，是内部使用的消息
quit()	关闭应用程序，这是内部使用的消息
quitIfInvisible()	如果应用程序未被显示，关闭应用程序。这是个内部使用的消息

QPE/Application/appname 信道有一个特殊的属性，当消息通过 QCopEnvelope 被发送到这些信道时，即使应用程序尚未运行，消息还是被分发。即不管应用程序是否运行，应用程序信道都是存在的。

5.3 国际化

5.3.1 翻译操作步骤

翻译 Qtopia 为所选语言的操作步骤如下。

（1）安装包括工具 linguist、lupdate 和 lrelease 在内的 Qt 3.x。

（2）运行 lupdate 和 lrelease，并开始使用 Trolltech 的 linguist 工具。

（3）使用 linguist 工具翻译字符串。

（4）加上语言描述。

（5）创建一个 ipkg 包。

运行命令 make lupdate 编译 lupdate 时，将创建目录:$QPEDIR/i18n/<new lang>和服务器特定的翻译文件（如包含 Qtopia 应用程序的翻译名字的 QtopiaApplications.ts），并且自动产生每种语言需要的.directory 文件。例如，在 $QPEDIR/i18n/de 中的 German.directory 开始部分如下：

```
[Translation]
File=QtopiaI18N
Context=German
[Desktop Entry]
Name[]=German
```

一旦完成了所有字符串的翻译，则可创建一个 ipkg 包。注意，对于某些语言（如中文或日文）。为了显示翻译的字符串，可能需要安装附加的字体文件在$QPEDIR/lib/fonts 文件中。

5.3.2 .qm 文件位置

Qtopia 软件的语言国际化与基于 Qt 的软件是一样的，字符串使用宏 tr() 。国际化的处理后产生 .qm 文件，该文件应如下放置：

```
/opt/Qtopia/i18n/language/appname.qm
```

appname 是应用程序运行的名字，language 是 ISO 639 语言代码，可跟随子集和一个 ISO 3166 本土语言，如 en、en_GB、ja 或 ja_JP。

对于作为共享库的插件来说，文件必须如下放置：

```
/opt/Qtopia/i18n/language/pluginname.qm
```

pluginname 是插件共享对象的文件名减去任何扩展名，但是如果插件不包括 lib 前缀，则将加上。

本地语言前缀的支持，例如，如果应用程序使用英国英语，则其有一个定义变化为美国英语的/opt/Qtopia/i18n/en_US/appname.qm 文件。带有本地语言前缀的翻译优先于不带这个前缀的翻译。

5.3.3 Desktop 文件国际化

Desktop 文件(如/opt/Qtopia/apps/Application/appname.desktop)也能被翻译,例如，texteditor.desktop 文件开始部分如下：

```
[Translation]
File=QtopiaApplications
Context=Notes
[Desktop Entry]
Comment[]=A Text Editing Program
Exec=textedit
Icon=TextEditor
Type=Application
MimeType=text/*
Name[]=Notes
```

这将引起/opt/Qtopia/i18n/language/QtopiaApplications.qm 文件被装载（如果尚未装载）并且在 Text Editor 上下文中带有“[]”的字符串被翻译。bin/nct_lupdate脚本被用来从.desktop 文件中维护 .ts 文件，这些文件都用 UTF-8编码。

5.3.4 文档国际化

应用程序的在线文档使用 Qt 的 richtext 窗口部件所支持的 HTML 子集编写，这样的文档文件必须如下放置。

（1）英语文档放置如下：

```
/opt/Qtopia/help/html/appname-suffixes.html
```

（2）非英语文件放置如下：

```
/opt/Qtopia/help/language/html/appname-suffixes.html
```

后缀-suffixes 是可选的，但必须以“ - ”开始来避免名字空间冲突，这些文件用 UTF-8 编码。

其他语言特定的文件如下。

- /opt/Qtopia/etc/dict/words-language ：语言的单词表
- /opt/Qtopia/etc/dict/words-language.dawg：单词表转换成 dawg 格式。
- /opt/Qtopia/etc/dict/common-language：语言常用的单词表，其输入法可能与不常用的单词发生冲突。
- /opt/Qtopia/etc/dict/common-language.dawg：常用的单词表转换成 dawg 格式。
- /opt/Qtopia/pics/pkim/dict-language-lc.png：image文件，是代表针对非本地讲话者的语言输入的文件。
- /opt/Qtopia/pics/pkim/dict-language-uc.png ： image文件，是代表大写输入的图标。

5.4 Qtopia 文件系统标准

Qtopia 软件装在目标系统的/opt/Qtopia 目录下,其中应用程序必须遵循表 5-7 所列文件的放置标准。

表 5-7 Qtopia 应用程序文件放置标准

文件类型	放置位置目录	说　明
可执行文件	/opt/Qtopia/bin/appname	可运行的程序
Image 文件	/opt/Qtopia/pics/appname/...	应用程序需要使用的任何 image 文件是标准且内建的 image 或这个应用程序定义的目录中的文件，可从 Resource 类中查看访问这些文件的说明
.desktop 文件	/opt/Qtopia/apps/category/appname.desktop	文件.desktop 定义在 launcher 和“开始”菜单中的入口
声音文件	/opt/Qtopia/sounds/appname/...	任何应用程序需要的声音文件应该是标准的内嵌声音，或应用程序特定目录下的文件
国际化文件	/opt/Qtopia/i18n/language/appname.qm	软件的翻译文件
英语在线文档	/opt/Qtopia/help/html/appname-suffixes.html	这些文件提供了软件的英语在线文档（后缀可选的），在线文档能引用在/opt/Qtopia/pics 目录下的 image 文件。这样可透明地引用图标和其他的 image 文件，而不需要拷贝这个文件
在线文档	/opt/Qtopia/help/html/appname-suffixes.html	这些文件是软件的特定语言下的在线文档（后缀可选），后缀由应用程序开发者选择。但 Qtopia 将自动地连接在窗口中的 Help 菜单到 appname-widgetname.html（如果这个文件存在），因此允许不同应用程序窗口有不同的帮助页
任务栏小应用程序	/opt/Qtopia/plugins/applets	任务栏小应用程序放在这个目录下
媒体播放编/解码文件	/opt/Qtopia/plugins/codecs	媒体播放器的编码解码程序放在这个目录下
字体	/opt/Qtopia/plugins/fontfactories	字体工厂扩展目录
输入法插件文件	/opt/Qtopia/plugins/inputmethods	输入法插件目录
网络建立文件	/opt/Qtopia/plugins/network	网络建立模块目录
配置文件	/opt/Qtopia/etc/default	配置文件应存在$HOME/Settings 下，通常由自动 Config 类的正常使用来实现

文件.desktop 定义了应用程序在 Launcher 和“开始”菜单中的入口，Launcher 是启动应用程序的操作界面。 category 是应用程序的类别，必须是下面类别中的一个，并且应该反映出希望应用程序出现在 Launcher 中的类别。

（1）Settings：**对用户环境产生影响的软件配置功能，这种软件从不创建文档。**

（2）Games：**仅用于娱乐目的的软件。**

（3）Applications：**所有其他软件。**

文件.desktop 格式类似于 Unix 桌面系统 KDE 和 Gnome 的配置文件格式，列出如下（＃表示注释）：

```
[Desktop Entry]
# 上面这行表示文件的开始
#执行文件的名字必须在/opt/Qtopia/bin 中
Exec=executablename
# 对于链接到应用程序来说，下面必须是 Application
Type=Application
#定义能在/opt/Qtopia/pics 找到的图标，自动检测后缀
#图标应至少是 32x32 且应该能从 14x14 到 48x48 平滑缩放
Icon=Iconfile
#定义这个应用程序能处理的 mimetype，多个 mimetype 之间用“，”分开。
MimeType=/audio/mp3
# 下面的几行定义了应用程序出现在 launcher 和“开始”菜单的名字
Name=My fancy application
#如果你想支持不同的语言
```

```
Name[de]=Meine Anwendung)
#整个文件用utf8编码
```

5.5 Qtopia 插件

Qtopia 插件分为应用程序插件、服务器插件和系统插件，应用程序插件是应用程序使用的插件：服务器插件是在启动时由 Qtopia 服务器进行装载的插件：系统插件是在应用程序启动时被装载的插件。分别说明如下。

（1）应用程序插件：有 Media Player(Music/Video)、Media Recorder、Today 和 Qtopia Desktop。

（2）服务器插件：有输入法（Input methods）、网络服务（Network services）和 Taskbar applets。

（3）系统插件：有 Font factories、Image codecs、Text codecs、Widget styles 和 Window frame styles。

5.5.1 建立插件的步骤

在 Qtopia 中通过类似 COM 的层来使用插件，建立一个插件的基本步骤如下。

（1）创建插件的接口

一个插件可以提供多个接口，但所有插件至少提供接口 QUnknownInterface。

IID_WidgetPlugin 为接口提供了一个惟一的 ID。下面是一个插件接口的例子，它提供一个插件名和一个创建的窗口部件：

```
// {05E0A4AB-DDC5-4449-85A9-828100DE00A9}
#ifndef IID_WidgetPlugin
#define IID_WidgetPlugin QUuid( 0x05e0a4ab, 0xddc5, 0x4449, 0x85, 0xa9, 0x82, 0x81, 0x00, 0xde, 0x00, 0xa9)
#endif

struct WidgetPluginInterface : public QUnknownInterface
{
   virtual QWidget *widget( QWidget *parent ) = 0;
   virtual QString name() const = 0;
};
```

（2）写插件将提供的功能函数

我们写的插件提供了一个窗口部件，可以画一个椭圆在其中心，代码如下：

```
class EllipseWidget : public QWidget
{
   Q_OBJECT
public:
   EllipseWidget( QWidget *parent=0 ) : QWidget( parent, "Ellipse" )
   {
   }

protected:
   void paintEvent( QPaintEvent * )
   {
      QPainter p( this );
      p.drawEllipse( rect() );
   }
};
```

（3）为接口定义子类

接口从 WidgetPluginInterface 派生的类，列出如下：

```
struct CirclePlugin : public WidgetPluginInterface
```

```
{
public:
   virtual QWidget *widget( QWidget *parent );
   virtual QString name() const;

   QRESULT queryInterface( const QUuid&, QUnknownInterface** );
   Q_REFCOUNT

protected:
   CircleWidget *w;
   ulong ref;
};
```

在 CirclePlugin 结构中需要注意如下两点。

- **queryInterface()函数允许插件装载器查询插件使用的接口。**
- **宏定义 Q_REFCOUNT 是一个简化的引用计数，这个宏使得引用计数变量出现。**

```
CirclePlugin::CirclePlugin()
   : w(0), ref(0)  //ref 必须初始化为 0
{
}

CirclePlugin::~CirclePlugin()
{
   delete w;
}
```

函数 *queryInterface()* 的样板代码如下：

```
QRESULT CirclePlugin::queryInterface( const QUuid &uuid, QUnknownInterface **iface )
{
   *iface = 0;
   if ( uuid == IID_QUnknown )
      *iface = this;
   else if ( uuid == IID_WidgetPlugin )
      *iface = this;
   else
      return QS_FALSE;

   (*iface)->addRef();
   return QS_OK;
}

//widget()函数返回 widget
QWidget *CirclePlugin::widget( QWidget *parent )
{
   if ( !w )
      w = new CircleWidget( parent );
   return w;
}

//函数 name()返回插件的名字
QString CirclePlugin::name()
{
   return qApp->translate( "WidgetPlugin", "Circle" );
}
```

（4）创建一个插件实例

必须使用下面的样板代码创建一个 widget插件的实例：

```
Q_EXPORT_INTERFACE()
{
```

```
    Q_CREATE_INSTANCE( CirclePlugin )
}
```

（5）装载插件

装载插件的推荐方法是使用类 PluginLoader，该类提供了简化枚举和插件的装载，并在安全模式启动系统时提供安全。插件装载器类期望插件安装在 $QPEDIR/plugins/type 目录中。

装载插件的方法列出如下：

```
PluginLoader loader( "Widgets" );
QStringList list = pluginLoader.list();
QStringList::Iterator it;
QValueList<WidgetPluginInterface*> widgetsList;
for ( it = list.begin(); it != list.end(); ++it ) {
    WidgetPluginInterface *iface = 0;
    if ( pluginLoader.queryInterface( *it, IID_WidgetPlugin,
            (QUnknownInterface**)&iface ) == QS_OK && iface ) {
        widgetsList.append( iface );
    }
}
```

5.5.2 装载插件的方法

装载插件方法如下。

（1）使用 PluginLoader 和 QLibrary 类

插件也是一种共享库，其装载使用类 PluginLoader和 QLibrary，下面分别说明。

类 PluginLoader 封装了插件的装载方法，其成员函数说明如表 5-8 所示。

表 5-8 类 PluginLoader 的成员函数及其说明

类成员函数名	功能说明
PluginLoader (const QString & type)	构造函数，创建 type 类型插件的插件装载器对象 PluginLoader。插件必须被安装在$QPEDIR/plugins/type 目录下
~PluginLoader ()	析构函数，释放对象 PluginLoader，注意这个函数不释放任何接口
clear ()	释放所有的接口
disabledList ()	返回已取消的插件名字的链表
bool nSafeMode () [static]	如果 Qtopia 当前在安全状态，返回 TRUE。在安全模式，list()将返回空链表且没有插件能被装载，从而允许不起作用的插件被取消
bool isEnabled (const QString & name)	如果插件 name 被激活时，返回 TRUE
list ()	返回可用的插件的链表
queryInterface (const QString & name, const QUuid & id, QUnknownInterface ** iface)	查询名为 name 的插件中 id 定义的接口。如果插件接口可用，iface 将装有这个接口的指针地址。如果找到接口，返回 QS_OK；否则返回 QS_FALSE
releaseInterface (QUnknownInterface * iface)	释放接口 iface，接口必须是由前面的 queryInterface 函数返回的接口
setEnabled (const QString & name, bool enabled = TRUE)	激活或取消名为 name 的插件，一个失效的插件可查询到，但不会被 list()函数返回

QLibrary 类提供了处理共享库的包裹器，它提供了共享库的装载 卸载、符号解析及接口查询等功能，这个类暂时从 Qt 3.0 拷贝过来。类 Qlibrary 的成员函数及其功能说明如表 5-9 所示，它使用 QLibrary::Policy 枚举类型定义了一个 QLibrary 被装载或卸载共享库的各种策略。这些策略类型说明如下。

- QLibrary::Delayed：共享库在需要使用时得到装载并在析构函数中被卸载。

- QLibrary::Immediately：共享库立即装载并在析构函数中被卸载。
- QLibrary::Manual：类似于 Delayed，库必须被手动卸载。

表 5-9 类 Qlibrary 的成员函数及其说明

类成员函数名称	功能说明
QLibrary (const QString & filename, Policy pol = Delayed)	构造函数，为共享库 filename 创建一个 QLibrary 对象。如果 pol 为 Immediately 类型，库立即装载。注意 filename 不需要包括平台相关的扩展名，如 windows 操作系统下的.dll；否则*.dll 库文件在 Linux 不识别，而应由 Qt 根据不同的平台来加上扩展名
~QLibrary ()	析构函数，删除 QLibrary 对象。如果策略不是手动的，这个库将不被卸载
bool isLoaded ()	返回库是否被装载
QString library ()	返回 QLibrary 对象处理的共享库的文件名，包括平台特定的扩展名，如 `QLibrary lib( "mylib" );` `QString str = lib.library();` 在 Windows 中返回 mylib.dll，在 Linux 中返回 libmylib.so
Policy policy ()	返回当前的策略类型
QRESULT queryInterface (const QUuid & request, QUnknownInterface ** iface)	向前传递查询给组件并返回查询结构，request 和 iface 传播到组件的 queryInterface 实现函数。如果需要，装载库。
void * resolve (const char * symb)	返回输出的字符 symb 的地址，如果需要，装载库。返回字符不能得到解析或者库装载失败，返回 NULL 下面的例子返回库解析的函数地址，并调用该函数： `typedef int (*addProc)( int, int );` `addProc add = (addProc) library->resolve( "add" );` `if ( add )` `  return add( 5, 8 );` `else` `  return 5 + 8;`
void * resolve (const QString & filename, const char * symb) [static]	这个函数是前面的函数的重载函数，为使用提供了方便，其功能相同。它装载库 filename，并返回输出字符 symb 的地址。注意 filename 不能包含平台相关的扩展名
void setPolicy (Policy pol)	设置策略 pol，如果 pol 设置为 Immediately 类型，则装载库
bool unload (bool force = FALSE)	释放组件并在组件释放成功时卸载库。如果库能被卸载，返回 TRUE；否则返回 FALSE。如果组件使用了 QLibraryInterface，这个接口的函数 cleanup()将被调用。如果随后调用了函数 canUnload()，卸载将被取消，并返回 FALSE。如果策略不是 Manual 类型，这个函数将在析构函数中被自动调用。如 force 被设置为 TRUE，库被强制卸载

（2）使用 PluginLoader 类实现

插件的配置文件必须放在 $QPEDIR/plugins 目录下，在安装拷贝后必须放在 $QTOPIA_PATH/plugins 目录下。每种应用类型插件对应一个子目录，如 applets 及 codecs 等。每种类型插件在子目录下用 .directory 配置文件描述这个类型的配置，每个插件使用 *.desktop 配置文件描述插件的配置，如 batteryapplet.desktop。这些配置在插件的装载过程中会使用到。

Qtopia 中插件的装载是通过类 PluginLoader 实现的，下面通过对这个类的分析来说明插件的装载过程。

PluginLoader 类构造函数 PluginLoader(const QString &type) 创建对象，并调用 PluginLoader 类成员函数 initType() 来初始化库文件名链表和插件名链表，列出如下：

```
PluginLoader::PluginLoader( const QString &type )
{
    d = new PluginLoaderPrivate;
```

```
    d->type = type;
    init();  //在多个应用程序下函数为空
    initType();
}
```

函数 initType()将某一类型插件（如 applet）目录中的库文件加入到链表中，从配置文件$HOME/Settings/Plugloader.conf 中读出插件的配置，如是否安全模式及未激活的插件等。同时，从库文件名中分离出插件名放入链表中。

函数 initType() 分析如下：

```
void PluginLoader::initType()
{
    QStringList qpepaths = global_qtopiapaths();  //得到Qtopia安装目录$QTOPIA_PATH
    for (QStringList::ConstIterator qit = qpepaths.begin(); qit!=qpepaths.end(); ++qit) {
        QString path = *qit + "plugins/"; //得到plugins路径，即：$QTOPIA_PATH/plugins
        path += d->type; //得到插件类型对应的目录，如$QTOPIA_PATH/plugins/applets

        DIR *dir = opendir( path.latin1() );  //打开目录文件
        if ( !dir )
            continue;

        QStringList list;
        dirent *file;
        while ( (file = readdir(dir)) ) {  //读取目录中每个文件名
            if ( !strncmp( file->d_name, "lib", 3 ) ) { //如果以文件名lib开头
                if ( !strcmp( file->d_name+strlen(file->d_name)-3, ".so" ) ) //文件名以.so结尾
                    list.append( file->d_name );  //将文件名加入到链表list中
            }
        }
        closedir(dir);

        bool safeMode = FALSE;

        //得到配置文件名，即$HOME/Settings/Plugloader.conf.lock
        QString cfgFilename( configFilename(cfgName()) + ".lock" );
        QFile lf( cfgFilename );
        //如果文件存在，仅读方式打开；否则仅写方式打开
        lf.open( lf.exists() ? IO_ReadOnly : IO_WriteOnly );
        lockFile( lf );  //锁住文件
        {
            Config cfg( cfgName() );  //创建配置文件$HOME/Settings/Plugloader.conf的Config对象
            cfg.setGroup( "Global" );  //设置组为Global
            //读出Global组中Mode成员值，默认值为Normal。如果值为"Safe"，safeMode=TRUE
            safeMode = cfg.readEntry( "Mode", "Normal" ) == "Safe";
            cfg.setGroup( d->type ); //组为插件类型，如applets
            //读出Disabled的值（','为分隔符）以放入字符链表中
            d->disabled = cfg.readListEntry( "Disabled", ',' );
        }
        unlockFile( lf );

        QStringList required;
        //如：$QTOPIA_PATH/plugins/applets /.directory存在
        if ( QFile::exists( path + "/.directory" ) ) {
            Config config( path + "/.directory", Config::File ); # Config::File表示绝对路径名
            required = config.readListEntry( "Required", ',' );
        }

        QStringList::Iterator it;
        for ( it = list.begin(); it != list.end(); ++it ) {
            //从库文件名得到插件名，如libbatteryapplet.so对应插件名为batteryapplet
            QString name = stripSystem(*it);
            //如果非安全模式且插件被激活或者required中含有name项
            if ( (!safeMode && isEnabled(name)) || required.contains(name) )
```

```
                d->plugins += name;//将插件名加入 plugins 中
            }
        }

    }
```

函数 queryInterface 利用类 QLibrary 查询名为 name 插件中 id 定义的插件接口，如果接口可用，将接口指针地址放在 iface 中。如果接口被发现，返回 QS_OK；否则返回 QS_FALSE。类 QLibrary 从插件共享库文件中得到库文件入口的字符表，从字符表中查询得到接口并保存在 iface 中。

函数 queryInterface 分析如下：

```
QRESULT PluginLoader::queryInterface( const QString &name, const QUuid &id, QUnknownInterface **iface )
{
    QRESULT result = QS_FALSE;
    *iface = 0;
    //从库文件名得到插件名，如 libbatteryapplet.so 对应插件名为 batteryapplet
    QString lname = stripSystem( name );

    QString libFile;
    QStringList qpepaths = global_qtopiapaths();//得到 Qtopia 安装目录$QTOPIA_PATH
    for (QStringList::ConstIterator qit = qpepaths.begin(); qit!=qpepaths.end(); ++qit) {
    //得到库文件名，如$QTOPIA_PATH/plugins/applets/libbatteryapplet.so
    libFile = *qit + "plugins/" + d->type + "/lib" + lname + ".so";

      if ( QFile::exists(libFile) )
         break;
    }

    //从 QLibrary 库对象链表 libs 中查找 libFile 对应的库，如未找到，创建 QLibrary 对象并加到 libs 中
    QLibrary *lib = LOADER_INSTANCE()->refLibrary( libFile );
    if ( !lib ) {
        qDebug( "Plugin not loaded: %s", lname.latin1() );
        return result;
    }

    bool enabled = isEnabled( name );
    if (enabled)  //如果插件激活
        setEnabled( name, FALSE );  //取消插件
    /*类 QLibrary 通过调用 dlopen( filename.latin1() , RTLD_LAZY )函数来打开共享库文件，读出库文件的入口表，每项以结构 QUnknownInterface 表示，这个结构包含接口字符指针地址 queryInterface、引用计数 addRef()及 release()。函数 queryInterface 查询 id 接口，并将找到的接口放入 iface 中*/
    if ( lib->queryInterface( id, iface ) == QS_OK && iface ) {
        d->interfaces.insert( *iface, lib );
        QString type = name;

        ......  //省略安装翻译文件
        result = QS_OK;
    } else {
        LOADER_INSTANCE()->derefLibrary( lib );
    }
    if (enabled)
        setEnabled( name, TRUE );  //激活插件

    return result;
}
```

5.6 输入法插件

Qtopia 2.x 支持两种输入法，即弹出输入法和复合输入法。弹出输入法在没有键盘和其他文件输入

设备时使用，它弹出窗口部件。用户可通过设备的触摸屏操作产生键事件，它们也被看做基于笔的输入方法；复合输入方法用来从独立的文本输入设备扩展或转换文本，这可能是一个键盘、小键盘或一个弹出输入法，称为“复合输入法”是因为它们在文体被窗口部件产生到被键盘焦点所在当前窗口接收之间提供了一个复合变换步骤。

创建这两种输入法插件的接口接口分别是 InputMethodInterface 和 ExtInputMethodInterface 类，InputMethodInterface类仅被用来创建弹出输入法,仅能访问 Qtopia 1.5.x 版本 API; ExtInputMethodInterface 类能用来创建这两种输入法。输入法的样例代码能在 Qtopia 源代码的 examples/inputmethods 目录下找到。

5.6.1 创建弹出输入方法

弹出输入法必须提供一个显示在屏幕上的 QWidget 窗口部件，并在一个键被按下时发出一个信号，Qtopia 处理输入窗口部件的显示和隐藏。可以在单个应用程序中开发一个弹出输入方法，并使用一个 QMultiLineEdit 窗口部件来测试文本的输出。在测试弹出输入方法的窗口部件后，需要把它转换为一个插件。

示例：简单弹出式输入法

下面是一个弹出输入方法示例，该输入方法是一个带有 A、B、C 和 D 键的虚拟键盘。

（1）为弹出输入方法设备窗口部件，弹出窗口部件的接口很简单，列出如下：

```
#include <qwidget.h>

class PopupIM : public QWidget
{
    Q_OBJECT
public:
    PopupIM(QWidget *parent, const char *name = 0, int flags = 0);
    ~PopupIM();

signals:
    void keyPress( ushort unicode, ushort keycode,
          ushort modifiers, bool press, bool repeat );
};
```

以后你将需要连接 keyPress 信号到 Qtopia，这个信号必须与将被提供的 slot 函数相匹配，keyPress 信号的参数说明如表 5-10 所示。

表 5-10 keyPress 信号的参数说明

参　数	说　明
unicode	字符的 unicode 值，或如果它是一个非字符键，其值为 0xFFFF
keycode	在 qnamespace.h 中定义的键编码
modifiers	Qt::ShiftButton，Qt::ControlButton 和 Qt::AltButton 的“或”逻辑运算值或 0
press	如果一个键被按下，返回 TRUE；否则返回 FALSE
repeat	如果按键重复时，返回 TRUE。当--个键被按下一段时间，重复按键作为附加事件发生

带有虚拟键盘的弹出输入法窗口部件 PopupIM 需要放一些虚拟键，虚拟键（实质上是按键控件）的 PopupPushButton 类列出如下：

```
class PopupPushButton : public QPushButton
{
```

```
    Q_OBJECT
public:
    PopupPushButton ( const QChar uni, ushort key, QWidget *parent )
        : QPushButton(parent), u(uni), k(key)
    {
        connect(this, SIGNAL(pressed()), this, SLOT(sendKeyForPress()));
        connect(this, SIGNAL(released()), this, SLOT(sendKeyForRelease()));
        setText(QChar(uni));
    }
signals:
    void keyPress(ushort, ushort, ushort, bool, bool);

private slots:
    void sendKeyForPress()
    {
        emit keyPress(u.unicode(), k, 0, TRUE, FALSE);
    }
    void sendKeyForRelease()
    {
        emit keyPress(u.unicode(), k, 0, FALSE, FALSE);
    }
private:
    QChar u;
    ushort k;
};
```

虚拟键 PopupPushButton 类能输入一个作为文本或键编码的字符，但不能增加 modifiers 或产生重复的键。

PopupIM 构造函数放置虚拟键在网格上，并将信号与其本身连接起来，列出如下：

```
PopupIM::PopupIM(QWidget *parent, const char *name, int flags)
    : QWidget(parent, name, flags)
{
    QGridLayout *gl = new QGridLayout(this, 2,2);
    PopupPushButton *ppb;

    ppb = new PopupPushButton(QChar('A'), Qt::Key_A, this);
    connect(ppb, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)),
            this, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)));
    gl->addWidget(ppb, 0, 0);
    ppb = new PopupPushButton(QChar('B'), Qt::Key_B, this);
    connect(ppb, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)),
            this, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)));
    gl->addWidget(ppb, 0, 1);
    ppb = new PopupPushButton(QChar('C'), Qt::Key_C, this);
    connect(ppb, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)),
            this, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)));
    gl->addWidget(ppb, 1, 0);
    ppb = new PopupPushButton(QChar('D'), Qt::Key_D, this);
    connect(ppb, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)),
            this, SIGNAL(keyPress(ushort, ushort, ushort, bool, bool)));
    gl->addWidget(ppb, 1, 1);
}
```

（2）创建一个 InputMethodInterface 插件

弹出输入方法使用 InputMethodInterface 转换到插件较简单，也可以使用 ExtInputMethodInterface 来转换。插件接口定义如下：

```
#include <qpe/inputmethodinterface.h>

class PopupIMImpl : public InputMethodInterface
{
```

```
public:
    PopupIMImpl();
    ~PopupIMImpl();

#ifndef QT_NO_COMPONENT
    QRESULT queryInterface( const QUuid&, QUnknownInterface** );
    Q_REFCOUNT
#endif

    QWidget *inputMethod( QWidget *parent, Qt::WFlags f );
    void resetState();
    QPixmap *icon();
    QString name();
    void onKeyPress( QObject *receiver, const char *slot );

private:
    PopupIM *input;
    QPixmap *icn;
    ulong ref;
};
```

插件接口的构造函数列出如下：

```
PopupIMImpl::PopupIMImpl()
    : input(0), icn(0), ref(0)
{
}

PopupIMImpl::~PopupIMImpl()
{
    if (input)
        delete input;
    if (icn)
        delete icn;
}
```

queryInterface() 函数的实现列出如下：

```
QRESULT PopupIMImpl::queryInterface( const QUuid &uuid, QUnknownInterface **iface )
{
    *iface = 0;
    if ( uuid == IID_QUnknown )
        *iface = this;
    else if ( uuid == IID_InputMethod )
        *iface = this;
    else
        return QS_FALSE;

    (*iface)->addRef();
    return QS_OK;
}
```

函数 inputMethod()返回输入法窗口部件，当用户需要输入文本时，这个窗口部件将显示在屏幕上。如果这个函数被调用多次，则应该总是返回同一个窗口部件。函数 inputMethod() 列出如下：

```
QWidget *PopupIMImpl::inputMethod( QWidget *parent, Qt::WFlags f )
{
    if ( !input )
        input = new PopupIM( parent, "SimpleInput", f );
    return input;
}
```

函数 resetState()应该返回输入法到其默认状态，因为 PopupIM 不需要保存状态，可以设置为空函数：

```
void PopupIMImpl::resetState() { }
```

函数 name()返回输入法的名字，这个名字将显示 在可用输入法的弹出菜单中。该函数列出如下：

```
QString PopupIMImpl::name()
{
    return qApp->translate( "InputMethods", "SimpleInput" );
}
```

函数 icon() 返回输入法的图标，当输入法被选中时，它显示在任务栏中。该函数列出如下：

```
QPixmap *PopupIMImpl::icon()
{
    if ( !icn )
        icn = new QPixmap( your pixmap );
    return icn;
}
```

函数 onKeyPress()是连接按键信号的槽函数，列出如下：

```
void PopupIMImpl::onKeyPress( QObject *receiver, const char *slot )
{
    if ( input )
        QObject::connect( input, SIGNAL(keyPress(ushort,ushort,ushort,bool,bool)), receiver, slot );
}
```

还必须创建输入法插件的实例，列出如下：

```
Q_EXPORT_INTERFACE()
{
    Q_CREATE_INSTANCE( PopupIMImpl )
}
```

（3）编译插件

为编译输入法插件，需要在.pro 文件中设置如下：

```
TARGET      = popupim

CONFIG     += qtopiaplugin
HEADERS     = popupim.h popupimpl.h
SOURCES     = popupim.cpp popupimpl.cpp

QTOPIA_PROJECT_TYPE=inputmethods
```

运行如下命令产生 Makefile 文件：

```
qmake -spec $QPEDIR/mkspecs/qws/linux-generic-g++ popup.pro
```

5.6.2 创建复合输入法

复合输入法从服务器过滤键事件，而不是产生键事件。当作为一个插件编译时， Qtopia 将增加或删除过滤器到服务器。

示例：简单的复合输入法

下面 列出了一个简单的复合输入法的实现例子，用来说明复合输入法控件的实现过程。

（1）为复合输入法设计过滤器

复合输入法的主类 ComposeIM 继承自 QWSInputMethod，它在事件被送到键盘焦点所在的应用程序之前过滤所有的键盘事件。ComposeIM 类列出如下：

```
class ComposeIM : public QWSInputMethod
{
public:
    ComposeIM();

    void reset();
    bool filter(int unicode, int keycode, int modifiers,
                            bool isPress, bool autoRepeat);

    enum State { Off, On };

private:
    State state;
    QString composed;
};
```

函数 QWSInputMethod::filter()是输入法的核心部件，它使用混合逻辑并维护状态，并使用 QWSInputMethod::sendIMEvent() 发送输入法事件。

当输入法需要重置状态时，系统调用函数 QWSInputMethod::reset()。例如焦点所在的窗口部件变化时，需要重置输入法状态。

当在焦点窗口部件内部的光标位置变化时，系统调用函数 QWSInputMethod::setMicroFocus()；当用户点击文本内部时，系统调用函数 QWSInputMethod::mouseHandler()。构造函数与析构函数列出如下：

```
ComposeIM::ComposeIM()
{
    state = Off;
}

ComposeIM::~ComposeIM()
{
}
```

函数 reset()设置 IM 到初始状态，列出如下：

```
void ComposeIM::reset()
{
    if ( state == On ) {
        state = Off;
        sendIMEvent( QWSServer::IMEnd, composed, 0 );
        composed = "";
    }
}
```

函数 filter()从服务器过滤按键事件，下面的代码提供了开/关输入法及文本的变换方法。这个例子使用 Shift+Space 联合键开/关输入法，另外还使用 compose 函数变换当前的文本。过滤函数 filter 返回 TRUE 来阻塞起始的按键事件，返回 FALSE 来使这个起始的按键事件传递到当前应用程序和焦点所在的窗口部件，并且使用函数 sendIMEvent 产生附加的输入事件。

函数 filter()列出如下：

```
bool ComposeIM::filter(int unicode, int keycode, int modifiers,
                        bool isPress, bool autoRepeat)
{
```

```
    if ( isPress && keycode == Qt::Key_Space &&
         modifiers & Qt::ShiftButton ) {
        //切换到相反的状态
        if ( state == On ) {
            sendIMEvent( QWSServer::IMEnd, QString::null, 0 );
            composed = "";
            state = Off; //重置并删除文本
        } else {
            state = On;
        }
        return TRUE; //阻塞事件
    } else if ( state == On ) {
        if ( isPress ) {
           if ( keycode == Qt::Key_Return ) {
               //接收文本并保持激活
               sendIMEvent( QWSServer::IMEnd, composed, composed.length() );
               composed = "";
           } else if ( keycode == Qt::Key_Backspace ) {
               if ( composed.length() > 0 )
                   composed = composed.left( composed.length() - 1 );
               sendIMEvent( QWSServer::IMCompose, composed, composed.length(), 0 );
           } else if ( unicode > 0 && unicode < 0xffff) {
               composed += QChar( unicode );
               //在送到输入窗口部件之前变换给定的文本
               composed = compose( composed );
               sendIMEvent( QWSServer::IMCompose, composed, composed.length(), 0 );
           }
       }
       return TRUE; //阻塞事件
    }
    return FALSE; //正常传递keystroke
}
```

（2）创建一个 ExtInputMethodInterface 插件

复合输入法在创建插件时必须使用 ExtInputMethodInterface 类，这个插件的接口列出如下：

```
class ComposeImpl : public ExtInputMethodInterface
{

public:
    ComposeImpl();
    virtual ~ComposeImpl();

#ifndef QT_NO_COMPONENT
    QRESULT queryInterface( const QUuid&, QUnknownInterface** );
    Q_REFCOUNT
#endif
    virtual QString name();
    virtual QPixmap *icon();

    virtual void resetState();

    virtual QStringList compatible();

    virtual QWSInputMethod *inputMethod( );

    virtual QWidget *statusWidget( QWidget *parent, Qt::WFlags f);
    virtual QWidget *keyboardWidget( QWidget *parent, Qt::WFlags f) { return 0; }

    virtual void qcopReceive( const QCString &msg, const QByteArray &data ) { }

private:
    ComposeIM *input;
    QPixmap *icn;
```

```
    QWidget *statWid;
    ulong ref;
};
```

函数 queryInterface()的实现如下：

```
QRESULT ComposeImpl::queryInterface( const QUuid &uuid, QUnknownInterface **iface )
{
    *iface = 0;
    if ( uuid == IID_QUnknown )
        *iface = this;
    else if ( uuid == IID_ExtInputMethod )
        *iface = this;
    else
        return QS_FALSE;

    (*iface)->addRef();
    return QS_OK;
}
```

函数 name()返回输入法的名字，放在可用输入法列表中，列出如下：

```
QString ComposeImpl::name()
{
    return qApp->translate( "InputMethods", "SimpleInput" );
}
```

函数 icon()返回输入法的图标，列出如下：

```
QPixmap *ComposeImpl::icon()
{
    if ( !icn )
        icn = new QPixmap( your pixmap );
    return icn;
}
```

函数 resetState() 重置输入法到初始状态，列出如下：

```
void ComposeImpl::resetState()
{
    if ( input )
        input->reset();
}
```

函数 compatible()用来说明这个输入法仅与其他某种输入法相适应。在这个例子中，返回空表示没有限制：

```
QStringList ComposeImpl::compatible( )
{
    return QStringList();
}
```

函数 inputMethod() 返回输入法的对象指针，列出如下：

```
QWSInputMethod *ComposeImpl::inputMethod( )
{
    if ( !input )
        input = new ComposeIM( );
    return input;
}
```

当输入法被选择后，函数 statusWidget()返回输入法窗口部件放在任务栏中。这个窗口部件用来显示状态，并允许用户与输入法进入交互。函数 statusWidget()列出如下：

```
QWidget *ComposeImpl::statusWidget( QWidget *parent, Qt::WFlags )
{
    if (!statWid) {
        (void) inputMethod(); //create input before we use it
        statWid  = new IMStatus( input, parent);
    }
    return statWid;
}
```

函数 keyboardWidget()对于复合输入法来说尚未应用，返回 0.

函数 qcopReceive()被调用来在输入法信道上通知插件事件，因为我们的插件不需要在输入法信道上说明输入法的任何事件，所以这个函数为空函数。

必须使用下面的代码产生一个输入法插件的实例：

```
Q_EXPORT_INTERFACE()
{
   Q_CREATE_INSTANCE( PopupIMImpl )
}
```

（3）编译插件

为了编译插件，必须如下设置.pro 文件：

```
TARGET      = composeim
CONFIG     += qtopiaplugin
HEADERS     = composeim.h composeimpl.h
SOURCES     = composeim.cpp composeimpl.cpp

QTOPIA_PROJECT_TYPE=inputmethods
```

使用如下命令创建 Makefile 文件：

```
qmake -spec $QPEDIR/mkspecs/qws/linux-generic-g++ composing.pro
```

5.7 快速启动应用程序

应用程序的启动时间在用户设备上是很重要的，它与启动时加载的代码大小有关。通常影响启动时间的一些因素有装载 / 分析配置文件、创建用户界面组件(如对话框等)和读取数据等。Qtopia 提供了改进应用程序启动时间的方法，本节说明这些方法。

5.7.1 改进主窗口创建时间

改进启动时间的最简单的方法是延迟某些操作直到 UI 界面可见。如果可能，在执行需要的操作时不创建对话框，直到实际需要时创建。

示例 1 ：在需要时创建对话框

下面这个样例代码在主窗口 MainWidget 的构造函数中不创建 settingsDialog 对话框，而是在需要的时候才创建：

```
MainWidget::MainWidget( QWidget *parent, const char *name, WFlags f )
```

```
    : QMainWindow( parent, name, f ), settingsDialog(0)
{
}

void MainWidget::showSettings()
{
    // 如果 settingsDialog 尚未创建，现在创建它
    if ( !settingsDialog )
        settingsDialog = new SettingsDialog( this );
    settingsDialog-&gt;exec();
}
```

示例 2：分步创建窗口

如果在应用程序可见后，操作是立即需要的，则在主窗口部件可见后使用定时器函数 QTimer::singleShot()**启动操作。这样分步创建主窗口，方法如下：**

```
MainWidget::MainWidget( QWidget *parent, const char *name, WFlags f )
    : QMainWindow( parent, name, f ), settingsDialog(0)
{
    // 在事件队列已被处理且主窗口已可见后装载数据
    QTimer::singleShot(0, this, SLOT(loadData()));
}

void MainWidget::loadData()
{
    // 打开数据文件并把数据传送室数据视图中
}
```

5.7.2 Quick Launcher

旧的 Qtopia **版本应用程序启动较慢的原因主要有装载和连接动态库、构造** QPE/Application **和窗口部件，以及装载数据文件。** Quick Launcher **（快速启动器）的目标是减少装载和连接动态库时间、消除构造** QPE/Application **时间并部分减少构造窗口部件的时间。**

Quick Launcher **通过将应用程序作为一个共享库运行的方法来减少启动时间，为此需要将应用程序的** main **主函数做特殊处理。设计的方法是创建一个** stub **（桩）应用程序，在被请求的应用程序之前运行，这样就减少了装载/连接时间和** QPEApplication **对象的构造时间。**

当这个 stub **应用程序被启动时，它将连接大部分通用库、构造** QPEApplication **对象循环、预装载默认字体及初始化其他可能需要的数据、执行事件循环并等待运行一个应用程序的请求。**

当这个 stub **应用程序被请求运行一个应用程序时，它将装载可快速启动的应用程序的库并运行应用程序的入口点函数，改变身份到请求的应用程序等。它还将被预定到** QPE/Application/exename **信道，并从** Quick Launcher **信道脱离，然后返回到事件循环中。这时，** stub **应用程序的运行看起来如同是通过正常的进程被启动的。**

Quick Launcher **是一种预装的** stub **应用程序，可装载插件并个性化插件应用程序。该应用程序运行后，将其他各个应用程序的主窗口对象及应用程序名存放在一个对象映射链表中，所有的应用程序都共用** Quick Launcher **应用程序的** QPEApplication **对象及主事件循环，因此省略了应用程序装载时间。**

在服务器进程（ qpe**）中，** Quick Launcher **是** AppLauncher **类的一部分。** AppLauncher **类对** Qtopia **应用程序的启动负责并监控其运行的状态，该类总有一个** stub **应用程序运行准备接收请求运行一个应用程序。**

当一个运行应用程序的请求被发出时，如果应用程序不是可快速启动的，则正常启动。如果一个

Stub应用程序尚未运行，则正常启动；否则 stub 应用程序将通过 QCop 信道发送来一个消息启动这个应用程序。

在一个短暂的延迟（2 秒）后，一个新的 Stub 应用程序将被启动来处理下一个请求。因为 Stub 应用程序是服务器 qpe 的子进程，所以 AppLauncher 将继续正常的功能等，捕获 SIGCHILD 信号并处理应用程序的退出。

如果服务器作为 root 用户运行，并且确保其不影响前台进程所需要的处理时间时，服务器将把 stub 应用程序设置为较低的优先级。当接收到请求装载一个新的应用程序的消息时， stub 应用程序将被设置正常的优先级。

5.7.3 应用 Quick Launcher

为了创建成为 Quick Launcher 应用程序插件，main.cpp 必须使用下面两个宏来提供应用程序的入口点的声明：

```
QTOPIA_ADD_APPLICATION(NAME,IMPLEMENTATION)
QTOPIA_MAIN
```

然后修改 qmake 项目文件作为一个库来编译应用程序并编译通过定义 QTOPIA_APP_INTERFACE 来编译插件接口，例如：

```
TEMPLATE = lib
CONFIG  += qtopia
DEFINES += QTOPIA_APP_INTERFACE
HEADERS  = mymainwindow.h
SOURCES  = mymainwindow.cpp main.cpp
TARGET   = myapp
```

应用程序插件必须安装在 QPEDIR/plugins/application/目录。

当编译一个 Quick Launcher 插件的应用程序时需要小心符号冲突，因为应用程序作为一个将被装载的插件编译，其中使用符号与库符号的冲突是可能的。

例如，如果有同一个名的两个类，一个在应用程序中并且另一个在应用程序链接的库中，那么当应用程序作为一个 Quick Launcher 应用程序运行时，这个类的错误析构函数将被调用。这可能引起内存崩溃。

脚本$QPEDIR/scripts/checksymbols 帮助识别可能的符号冲突，确信 QPEDIR 定义后运行这个脚本，传递应用程序创建的.so 文件作为命令行参数。这个脚本将列出在应用程序和 Qtopia 库之间可能的符号冲突。

5.7.4 Quick Launcher 的宏定义

Quick Launcher 使用的两个宏说明如下。

（1）宏定义 QTOPIA_ADD_APPLICATION()：产生主窗口实例化的代码，NAME 是应用程序执行名字，IMPLEMENTATION 是应用程序的窗口类。

（2）宏定义 QTOPIA_MAIN：main()的应用或代码使用应用程序成为一个 Quick Launcher 插件需要的接口（如果 QTOPIA_APP_INTERFACE 被定义）。

应用程序不应该执行应用程序级操作（如测试参数及调用 exit()或 quit()），所有这些应被 QPEApplication 处理。应用程序窗口如果明确需要关闭，则应该调用 close()。

下面对这两个宏定义的源代码进行分析。

宏定义 QTOPIA_ADD_APPLICATION(在 src/libraries/qtopia/qpeapplication.h 中)创建主窗口对象，

并将应用程序名及主窗口对象加入到对象映射链表＊qpeAppMap()中，该宏定义 QTOPIA_ADD APPLICATION 分析如下：

```
typedef QWidget* (*qpeAppCreateFunc)(QWidget*,const char *,Qt::WFlags);
typedef QMap<QString,qpeAppCreateFunc> QPEAppMap;

#define QTOPIA_ADD_APPLICATION(NAME,IMPLEMENTATION) \
   static QWidget *create_ ## IMPLEMENTATION( QWidget *p, const char *n, Qt::WFlags f ) { \
     return new IMPLEMENTATION(p,n,f); } \        //创建 IMPLEMENTATION 主窗口对象
   QPEAppMap *qpeAppMap(); \
   static QPEAppMap::Iterator dummy_ ## IMPLEMENTATION = qpeAppMap()->insert(NAME,create_ ## IMPLEMENTATION); //插入到对象映射链表中
```

宏定义 QTOPIA_MAIN 根据 3种不同情况运行应用程序 main()主函数，下面分别说明这 3 种情况。

（1）在宏定义 QTOPIA_NO_MAIN情况下

宏定义 QTOPIA_MAIN 为空，没有 main() 主函数。

（2）在宏定义 QTOPIA_APP_INTERFACE 情况下

宏定义 QTOPIA_MAIN 创建对象接口 ApplicationImpl ,对象接口 ApplicationImpl 中的接口函数创建主窗口对象，并将主窗口对象和应用程序名加入到对象映射链表中。这时宏定义 QTOPIA_MAIN 只是完成插件的功能，这种情况用于快速启动器。

（3）在其他情况下

宏定义 QTOPIA_MAIN 的操作与正常情况下 Qtopia 应用程序 main()主函数的调用函数一样。

宏定义 QTOPIA_MAIN（在 src/libraries/qtopia/qpeapplication.h 中）分析如下：

```
#ifdef QTOPIA_NO_MAIN
#define QTOPIA_MAIN

#else

#ifdef QTOPIA_APP_INTERFACE

# define QTOPIA_MAIN \
    struct ApplicationImpl : public ApplicationInterface { \
        ApplicationImpl() : ref(0) {} \        //引用计数赋 0
        QRESULT queryInterface( const QUuid &uuid, QUnknownInterface **iface ) { \  //查询接口
            *iface = 0; \
            if ( uuid == IID_QUnknown ) *iface = this; \              //得到接口地址
            else if ( uuid == IID_QtopiaApplication ) *iface = this; \
            else return QS_FALSE; \
            (*iface)->addRef(); \    //引用计数加 1
            return QS_OK; \
        } \
        //函数*createMainWindow 从对象映射链表 qpeAppMap()得到主窗口对象，并返回对象
        virtual QWidget *createMainWindow( const QString &appName, QWidget *parent, const char *name, Qt::WFlags f ) { \          //创建主窗口
            if ( qpeAppMap()->contains(appName) ) \    //对象链表中含有 appName 的对象
                return (*qpeAppMap())[appName](parent, name, f); \    //返回对象
            return 0; \
        } \
        //这个函数从对象映射链表 qpeAppMap()得到所有应用程序名字的字符串链表
        virtual QStringList applications() const { \
            QStringList list; \
            for ( QPEAppMap::Iterator it=qpeAppMap()->begin(); it!=qpeAppMap()->end(); ++it ) \
                list += it.key(); \
            return list; \
        } \
        Q_REFCOUNT \    //引用计数加 1
        private: \
            ulong ref; \
```

```
    }; \
    QPEAppMap *qpeAppMap() { \    // 创建或得到静态的对象映射链表，用于存储主窗口对象及应用程序名
        static QPEAppMap *am = 0; \
        if ( !am ) am = new QPEAppMap(); \
        return am; \
    } \
    //创建 ApplicationImpl 对象，并返回接口对象地址
    Q_EXPORT_INTERFACE() { Q_CREATE_INSTANCE( ApplicationImpl ) }

#else   //其他情况下

# define QTOPIA_MAIN \    //这个 main() 函数与一般的 Qtopia 应用程序的 main() 函数一样
    QPEAppMap *qpeAppMap(); \
    int main( int argc, char ** argv ) { \
        QPEApplication a( argc, argv ); \    //创建 QPEApplication 对象 a
        QWidget *mw = 0; \
\
    QString executableName(argv[0]); \
    executableName = executableName.right(executableName.length() \
            - executableName.findRev('/') - 1); \
\
        //通过应用程序名从对象映射链表中得到主窗口对象
        if ( qpeAppMap()->contains(executableName) ) \
            mw = (*qpeAppMap())[executableName](0,0,0); \
        else if ( qpeAppMap()->count() ) \    //如果从对象映射链表中没找到，得到第 1 个主窗口对象
            mw = qpeAppMap()->begin().data()(0,0,0); \
        if ( mw ) { \ //如果主窗口对象存在
            if ( mw->metaObject()->slotNames(true).contains("setDocument(const QString&)") ) \
                a.showMainDocumentWidget( mw ); \  //显示主文档窗口对象
            else \
                a.showMainWidget( mw ); \  //  显示主窗口对象
            int rv = a.exec(); \  // 进入主事件循环
            delete mw; \
            return rv; \
        ) else { \
            return -1; \
        } \
    } \
    QPEAppMap *qpeAppMap() { \  //创建或得到静态的对象映射链表，用于存储主窗口对象及应用程序名
        static QPEAppMap *am = 0; \
        if ( !am ) am = new QPEAppMap(); \
        return am; \
    } \

#endif
#endif
```

宏定义 Q_CREATE_INSTANCE（在 src/libraries/qtopia/qcom.h 中）创建 IMPLEMENTATION 对象，并返回接口对象地址，列出如下：

```
#define Q_CREATE_INSTANCE( IMPLEMENTATION )       \
        IMPLEMENTATION *i = new IMPLEMENTATION; \
        QUnknownInterface* iface = 0;             \
        i->queryInterface( IID_QUnknown, &iface );     \
        return iface;
```

应用程序接口 ApplicationInterface（在 src/libraries/qtopia/applicationinterface.h 中）定义了一个应用程序的接口，这个接口可以创建主窗口对象，并得到应用程序名字符串链表，其定义如下：

```
struct ApplicationInterface : public QUnknownInterface
{
public:
    //创建主窗口对象
```

```
        virtual QWidget *createMainWindow( const QString &appName, QWidget *parent=0, const char *name=0,
Qt::WFlags f=0 ) = 0;
        virtual QStringList applications() const = 0;  //得到应用程序名字符串链表
    };
```

5.7.5 运行 Quick Launcher 分析

本节分析 Quick Launcher 的运行。

（1）Quick Launcher应用程序启动过程

Qtopia在嵌入设备启动时创建 Server 服务器对象，Server类构造函数会在创建应用程序管理窗口界面 UI 后创建 AppLauncher 对象，AppLauncher 类构造函数在延时 100 ms 后调用函数 createQuickLauncher() 来启动 Quick Launcher 应用程序。

Server 类构造函数与创建 AppLauncher 对象相关的代码列出如下：

```
Server::Server() :
    QWidget( 0, 0, WStyle_Tool | WStyle_Customize ),
    qcopBridge( 0 ),
    transferServer( 0 ),
    packageHandler( 0 ),
    syncDialog( 0 ),
    soundserver( 0 ),
    qrr( 0 ),
    lastStartedApp( 0 )
{
    ……
#ifdef QTOPIA_PHONE
    serverGui = new PhoneImpl;
#else
    serverGui = new Launcher;
#endif
    serverGui->createGUI();    //创建应用程序管理窗口 UI

    docList = new DocumentList( serverGui );
    appLauncher = new AppLauncher(this);
    ……
}
```

AppLauncher类构造函数调用函数 createQuickLauncher 的相关代码列出如下：

```
AppLauncher::AppLauncher(QObject *parent, const char *name)
    : QObject(parent, name), qlProc(0), qlPid(0), qlReady(FALSE),
      appKillerBox(0)
{
    ……
    QTimer::singleShot( 1000, this, SLOT(createQuickLauncher()) );
}
```

函数 createQuickLauncher 启动 quicklauncher应用程序，代码列出如下：

```
void AppLauncher::createQuickLauncher()
{
    qlReady = FALSE;
    //类 QPress 用来启动外部应用程序并与其通信
    qlProc = new QProcess(this);
    //设置 QProcess 进程通信标识（即设置是否传入或传出数据），默认为 Stdin|Stdout|Stderr
    qlProc->setCommunication(0);
    //当外部应用程序进程退出时连接处理函数 processExited()
    connect(qlProc, SIGNAL(processExited()), this, SLOT(processExited()));
```

```
    //追加参数到进程运行的参数链表中，链表中的第 1 个参数是执行的应用程序的名字
    //即应用程序$ QTOPIA_PATH/bin/quicklauncher
    qlProc->addArgument(QPEApplication::qpeDir()+"bin/quicklauncher");
    setenv( "LD_BIND_NOW", "1", 1 );  //设置环境变量
    bool started = qlProc->start();  //启动应用程序 quicklauncher
    if (!started) {//  启动不成功，再次启动
        qlProc->clearArguments();
        qlProc->addArgument("quicklauncher");
        started = qlProc->start();
    }
    if (started) {//  启动成功
        qlPid = (int)qlProc->processIdentifier();//    得到启动的应用程序的 pid
        if (getuid() == 0)
        setpriority( PRIO_PROCESS, qlPid, 19 );//设置运行优先级
    } else {  //启动不成功，删除创建的对象实例
        delete qlProc;
        qlProc = 0;
        qlPid = 0;
    }
}
```

（2）Quick Launcher 应用程序分析

Quick Launcher 应用程序启动后一直运行在设备中，当用户在应用程序管理窗口界面 UI 中点击应用程序图标时，会发送消息到 Quick Launcher 应用程序中。该应用程序处理这个消息，在点击应用程序的库中创建这个应用程序的主窗口，并回到应用程序事件主循环，这样用户感觉如同重新启动了一个应用程序。

Quick Launcher 应用程序主函数 main 创建 QPEApplication 对象，处理应用程序参数及环境变量后创建 Quick Launcher 对象，进入主事件循环。

下面按 Quick Launcher 应用程序的启动及在应用程序管理窗口界面 UI 上点击应用程序图标时的函数调用顺序来分析 Quick Launcher 类的成员函数。

构造函数 Quick Launcher()列出如下：

```
Quick Launcher() : QObject()
{
        QCString ch("QPE/QuickLauncher-");
        ch += QString::number(getpid());
        qlChannel = new QCopChannel( ch, this);
        connect( qlChannel, SIGNAL(received(const QCString&,const QByteArray&)),
                 this, SLOT(message(const QCString&,const QByteArray&)) );
}
```

函数 message是槽函数，它根据接收到的消息来启动被点击的外部应用程序窗口，分析如下：

```
void message(const QCString &msg, const QByteArray & data)
    {
       QStrList argList;

       if ( msg == "execute(QStrList)" ) {
           delete qlChannel;
           QDataStream stream( data, IO_ReadOnly );
           QStrList argList;
           stream >> argList;
           qDebug( "QuickLauncher execute: %s", argList.at(0) );
           doQuickLaunch( argList );
           delete this;
       } else if ( msg == "execute(QString)" ) {
           delete qlChannel;
```

```
            QDataStream stream( data, IO_ReadOnly );
            QString arg;
            stream >> arg;
            qDebug( "QuickLauncher execute: %s", arg.latin1() );
            QStrList argList;
            argList.append( arg.utf8() );
            doQuickLaunch( argList );
            delete this;
        }
    }
```

函数 doQuickLaunch 改变应用程序进程名，回到主事件循环并建立应用程序信道等，分析如下：

```
void doQuickLaunch( QStrList &argList )
    {
       static int myargc = argList.count()   ;
       static char **myargv = new char *[myargc + 1];
       for ( int j = 0; j < myargc; j++ ) {
           myargv[j] = new char [strlen(argList.at(j))+1];
           strcpy( myargv[j], argList.at(j) );
       }
       myargv[myargc] = NULL;
#ifdef _OS_LINUX_
       // 改变进程的名字，从 Quick Launcher 改变为在 UI中点击的应用程序的名字
       setproctitle(myargv[0]);  // myargv[0]为在应用程序管理窗口界面 UI中点击的应用程序的名字
#endif
       //app是 main 函数中创建的 QPEApplication 实例
       connect(app, SIGNAL(lastWindowClosed()), app, SLOT(hideOrQuit()));
       validExitLoop = TRUE;
       app->exit_loop();// 从递归的调用中到主事件循环
       app->initApp( myargc, myargv );  // 建立应用程序名信道，重载应用程序的参数链表
       exec( myargc, myargv );
}
```

函数 exec 查找应用程序对应库的接口，并调用接口的 createMainWindow()函数来创建应用程序主窗口并显示，分析如下：

```
static void exec( int /*argc*/, char **argv )
   {
       QString appName = argv[0];
       int sep = appName.findRev( '/' );
       if ( sep > 0 )
          appName = appName.mid( sep+1 );

       appIface = 0;
       //从 appName 名字对应的应用程序库中查找接口
       if ( loader->queryInterface(appName, IID_QtopiaApplication, (QUnknownInterface**)&appIface) ==
QS_OK ) {
          mainWindow = appIface->createMainWindow( appName );  //调用接口的函数创建对象
       }
       if ( mainWindow ) {  //对象创建成功，显示主窗口
          if ( mainWindow->metaObject()->slotNames(true).contains("setDocument(const QString&)") ) {
            app->showMainDocumentWidget( mainWindow );
       } else {
          app->showMainWidget( mainWindow );
        }
     } else {//创建失败退出
        qWarning( "Could not create application main window" );
        exit(-1);
     }
   }
```

5.8 Qtopia 核心类库说明

5.8.1 QPEApplication 类

QPEApplication 类提供各种系统服务，这些服务对于所有的 Qtopia应用程序是可用的。通过使用 QPEApplication 替代 QApplication，一个标准的 Qt 应用程序变成一个 Qtopia 应用程序。它自动遵循风格变化、退出和显示窗口上升，在有文档产生的应用程序中，按照环境改变当前显示的文档。

应该使用函数 showMainDocumentWidget() 创建产生文档的应用程序，使用函数 showMainWidget() 创建非文档的应用程序。keepRunning() 函数指示在处理最后一个 QCop 消息后，应用程序是否继续运行，为此可通过 setKeepRunning()来改变设置。

当某种事件发生时，会发出相应的各种信号，例如 timeChanged()、clockChanged()、weekChanged()、dateFormatChanged()和 volumeChanged()。如果应用程序在应用程序的 QPE/Application/appname 信道上接收到 QCop 消息，则发出 appMessage() 信号。当同步开始和结束时，相应地发射 flush()和 reload() 信号。依赖于这些信号，应用程序应该存储和重装载任何参与同步的数据文件，这些信号大多数开始被接收并且未通过 appMessage()信号过滤。

QPEApplication 类还提供了一套有用的静态函数，函数 qpeDir()和 documentDir()返回相应的路径，函数 grabKeyboard()和 ungrabKeyboard()用来控制应用程序是否控制设备的物理按钮（如应用程序 launch 键）。函数 setStylusOperation() 设置触笔模式，函数 stylusOperation() 得到触笔模式。函数 setInputMethodHint()设置特定输入方法，函数 inputMethodHint()得到输入方法。

QPEApplication 类构造函数 QPEApplication() 构建一个应用程序对象，对于应用程序来说，参数 t 默认情况下是 GuiClient，只有 Qtopia 服务器参数 t 设置为 GuiServer。构造函数 QPEApplication()设置临时文件夹、安装消息处理句柄、装载字体与图片的编/解码库、设置应用程序字体及图标、建立系统信道、应用程序信道、安装翻译器、设置应用程序风格，并且安装事件过滤器等。

构造函数 QPEApplication() 列出如下：

```
QPEApplication::QPEApplication( int& argc, char **argv, Type t )
    : QApplication( hack(argc), argv, t )
#if defined(Q_WS_QWS) && !defined(QT_NO_COP)
, pidChannel(0)
#endif
#if defined(QTOPIA_TEST)
, hasPerfMonitor(0)
#endif
{
#if defined (_WS_QWS_)
    //PDA 和手机有不确定的输入，只要有编辑框就应该看见口令文本
    qt_lineedit_password_visible_on_focus = TRUE;
#endif

#if defined(Q_WS_QWS) && defined(Q_OS_UNIX)
    QString dataDir(Global::tempDir());   //得到临时文件夹路径字符串
    if ( mkdir( dataDir.latin1(), 0700 ) ) {  //创建临时文件夹
        ...
    }
    ......
#endif
    qInstallMsgHandler(qtopiaMsgHandler); //安装消息处理句柄，将信息输出到 stderr 标准文件中

#ifdef QTOPIA_PHONE
    //得到默认按键配置文件
    Config config( qtopia_internal_defaultButtonsFile(), Config::File );
```

```
    config.setGroup( "Device" ) ;
    QString pi = config.readEntry( "PrimaryInput", "Keypad" ).lower()   ;
    //非触摸屏意味着键盘模式编辑被激活
    qt_modalEditingEnabled = pi != "touchscreen"  ;
    mousePreferred = !qt_modalEditingEnabled;
    //打印出调试信息
    qDebug( "Modal editing is %s", qt_modalEditingEnabled ? "ENABLED" : "DISABLED" );
#endif

    QPixmapCache::setCacheLimit(256); // 小设备位图缓存限制为 256 KB
    d = new QPEApplicationData; //初始化主窗口一些信息参数
#ifdef Q_WS_QWS
    PluginLoaderIntern::init(); //初始化插件装载器
    d->loadTextCodecs();    //装载字体编/解码库
    d->loadImageCodecs(); //装载 Image 图片编/解码库
#endif

    int dw = desktop()->width(); //得到根窗口的宽度
    if ( dw < 200 ) {  //根据根窗口的宽度设置字体及大小图标的尺寸
        setFont( QFont( "helvetica", 8 ) ) ;
#ifdef Q_WS_QWS
# ifdef QTOPIA_PHONE
            AppLnk::setSmallIconSize(16); //设置小图标大小
# else
            AppLnk::setSmallIconSize(10);
# endif
            AppLnk::setBigIconSize(28);
#endif
    }

#ifdef QTOPIA_PHONE
    setFont( QFont( "helvetica", 14, QFont::Normal ) ); //设置字体
#endif
    /*设置默认的 MIME 类工厂为 ResourceMimeFactory，它设置帮助文件路径和支持的文件类型扩展为 "html" */
    QMimeSourceFactory::setDefaultFactory(new ResourceMimeFactory);

    connect(this, SIGNAL(lastWindowClosed()), this, SLOT(hideOrQuit()))   ;

#if defined(Q_WS_QWS) && !defined(QT_NO_COP)
    //创建系统信道，并连接槽 systemMessage
    sysChannel = new QCopChannel( "QPE/System", this );
    connect( sysChannel, SIGNAL(received(const QCString&,const QByteArray&)),
         this, SLOT(systemMessage(const QCString&,const QByteArray&)) );
    //创建名为应用程序名的信道，连接槽函数 plnMessage
    initApp( argc, argv );

#endif
#if defined(Q_WS_QWS)
    FontDatabase::loadRenderers(); //装载字体工厂插件
#endif

#ifndef QT_NO_TRANSLATION
    QStringList qms;
    qms << "qt";
    qms << "libqpe";
    qms << "libqtopia" ;
    qms << "libqtopia2" ;
    qms << d->appName;

    loadTranslations(qms); //安装翻译器
#endif

    applyStyle(); //从配置文件中读入风格 (Style) 相关数据，并生成相应风格

#ifndef QTOPIA_DESKTOP
```

```
    if ( type() == GuiServer ) {  //服务器
        setScreenSaverInterval(-1);
        setVolume();  //设置音量
#ifdef Q_WS_QWS
        QWSServer::setScreenSaver(new QPEScreenSaver);//设置屏幕保护
#endif
    }
#endif

    installEventFilter( this );  //安装事件过滤器

#ifdef Q_WS_QWS
    QPEMenuToolFocusManager::initialize();  //初始化菜单工具焦点管理器
#endif

#ifdef QT_NO_QWS_CURSOR
    // 如果没有鼠标箭头，就不需要tooltip
    QToolTip::setEnabled( FALSE );
#endif

}
```

5.8.2 应用程序配置与连接类

在应用程序中，config 类用来存储应用程序的配置信息， AppLnk 和 DocLnk 类反映服务器窗口中图标与应用程序或文档的连接关系， FileManager 类辅助 AppLnk 的文档的输入/输出。下面对这些类分别进行说明。

（1） Config 类

Config 类用来处理应用程序配置信息的存储与读取，其构造函数得到或创建一个配置文件中的配置信息，配置文件扩展名为 .conf。类成员函数 setGroup 设置组（group）的名字，利用类成员函数 readEntry 读出配置项的值，利用成员函数 writeEntry 写入配置项的值到配置文件。

Config 类提供了存储应用程序状态的功能，仅当你不希望其他人能改变状态时才应用保护一个 Config 存在。Config 类的成员函数及其功能说明如表 5-11 所示。

Config::Config(const QString &name,Domain domain) 构造函数创建一个配置对象，它可以装载或创建一个给定域 domain 中给定名字 name 的配置。其中参数 domain 取值为 File 或 User，File 表示配置文件名为参数 name，User 表示配置文件名为 homeDirPath + “/Settings” + “/” +name + “.conf” 。参数 domain 默认值是 User，是用户定义的配置。参数 name 中不能含有 “/” ，且应该是惟一的。参数 domain 为 File 时，参数 name 是一个绝对路径名。

一个配置文件中有许多配置项，一个配置项分别用 group （组）、entry（条目）和 value（值）来描述，举例说明如下：

```
[test]
testname=abc;
test2=efg;
```

在这个配置项中，group 为值为 test，entry 为 testname，value 为 abc。

表 5-11 Config类的成员函数及其功能说明

成员函数名	功能说明
allGroups()	返回配置对象 Config 中所有 group 的链表
clearGroup()	从当前 group 中删除所有的条目
hasKey (const QString & key)	如果当前组有一个名为 key 的条目，返回 TRUE；否则返回 FALSE

续表

成员函数名	功能说明
isValid ()	如果 Config 是一个有效的状态，返回 TRUE；否则返回 FALSE
operator!= (const Config & other)	如果两个配置 Config 对象是同一文件的引用，测试二者之间是否不等
operator== (const Config & other)	如果两个配置 Config 对象是同一文件的引用，测试二者之间是否相等
readBoolEntry (const QString & key, bool deflt = FALSE)	返回使用参数 key 存储的值（格式为 boolean 逻辑型），如果没有给定 key 的条目，返回 deflt 值
readByteArrayEntry (const QString & key)	返回参数 key 存储的值（格式为 QByteArray），如果没有发现，返回一个空数组
readByteArrayEntry (const QString & key, const QByteArray & dflt)	返回参数 key 存储的值（格式为 QByteArray），如果没有发现，返回 dflt 值
eadEntry (const QString & key, const QString & deflt = QString::null)	返回参数 key 存储的值（格式为字符串），如果没有，返回 deflt 值
readEntryCrypt (const QString & key, const QString & deflt = QString::null)	为 key 存储的加密条目返回解密后的字符串，如果未找到给定 key 的条目，返回 deflt
readListEntry (const QString & key, const QChar & sep)	返回存储在 key 上的字符串链表，分隔符为 sep
readListEntry (const QString & key)	返回保存在 key 中的字符串链表
readNumEntry (const QString & key, int deflt = -1)	返回存储在 key 中的整数条目，未找到 key 时返回 deflt
removeEntry (const QString & key)	从当前组中删除 key 条目，如果没有 key 条目，不执行任何操作
removeGroup()	删除当前组及其所有条目
removeGroup (const QString & g)	删除名为 g 的组及其所有条目
setGroup (const QString & gname)	设置当前组名称为 gname
timeStamp (const QString & name, Domain domain = User) [static]	返回名为 name 的配置文件的时间戳，它是个静态函数
writeEntry (const QString & key, const char * value)	写(key, value)条目到当前的组中
writeEntry (const QString & key, const QString & value)	写(key, value)条目到当前的组中
writeEntry (const QString & key, int num)	写(key, num)条目到当前的组中
writeEntry (const QString & key, const QStringList & lst, const QChar & sep)	写(key, lst)条目到当前的组中，字符串之间以 sep 为分隔符
writeEntry (const QString & key, const QStringList & lst)	写(key, lst)条目到当前的组中，字符串之间以^e 为分隔符。字符串中有^时，^将被^^替代，这样字符串可以含有包括^在内的各种字符了。空字符也是被允许的，在字符串中以^0 来记录
writeEntry (const QString & key, const QByteArray & byteArray)	写(key, byteArray)条目到当前组中，byteArray 以 base64 编码字符串
writeEntryCrypt (const QString & key, const QString & value)	写一个弱加密(key, value)条目到当前组中

（2）AppLnkSet 和 AppLnk 类

AppLnkSet 类用来管理应用程序的 AppLnk 对象集合，其构造函数创建代表给定目录下所有应用程序的 AppLnk 对象，AppLnkSet 类的成员函数及其功能说明如表 5-12 所示。

表 5-12 AppLnkSet 类的成员函数及其功能说明

成员函数名	功能说明
add (AppLnk * f)	增加 f 到 AppLnkSet 中，AppLnkSet 负责删除 f
children ()	返回 AppLnkSet 的成员链表
clear ()	返回 AppLnkSet 的成员链表
detachChildren ()	从 AppLnkSet 中分离所有 AppLnk 对象，AppLnkSet 变成空的，调用者负责删除 AppLnk 对象
find (int id)	返回给定 id 的 AppLnk 对象
findExec (const QString & exec)	返回给定 exec 属性的 AppLnk 对象
remove (AppLnk * f)	移去 f 到 AppLnkSet 中。调用者变为负责删除 f。如果 f 在 AppLnkSet 中，返回 TRUE；否则返回 FALSE
typeBigPixmap (const QString & t)	返回与类型 t 相关的大图标，对于应用程序、游戏和设置来说，类型 t 是应用程序；对于文档来说，类型是文件的 MIME 类型
typeName (const QString & t)	返回类型 t 的本地化名字
typePixmap (const QString & t)	返回与类型 t 相关的小图标位图
types ()	返回 AppLnkSet 中类型的链表

AppLnk 类代表系统中一个可用的应用程序，每个 Qtopia 应用程序（如名为“ app ”的应用程序）有一个相应的 app.desktop 文件，这个文件描述了应用程序的名字、类型、显示名字及图标等信息。

当读取.desktop 文件时，文件的数据存在一个 AppLnk 对象中。程序在处理读取应用程序的信息数据时，只需要与 AppLnk 对象打交道。而与.desktop 文件打交道的工作交给 AppLnk 类处理。这样的分层会提高执行效率及程序的透明性，即应用程序不必关心.desktop 文件存储在何处。应用程序信息管理的流程如图 5-4 所示。



图 5-4 应用程序信息管理的流程

类 AppLnkSet 是 AppLnk 对象的集合，它提供了 AppLnk 对象链表中对对象的管理。

AppLnk 对象通过调用带有一个.desktop 文件名的构造函数来创建，AppLnk 类包括一些 Qtopia 特定的概念，如类型、文件与连接及功能等。分别说明如下。

（1）类 型

每个 AppLnk 对象有一个类型，对于应用程序、游戏和设置来说，类型是应用程序；对于文档来说，是文档的 MIME 类型。

（2）文件和连接

当创建一个 AppLnk 对象时，不能直接与文件系统中的文件名打交道，而应该如下操作：

```
DocLnk d;
d.setType("text/plain");
d.setName("My Nicely Named Document / Whatever");   // "/" 是合法的
```

在这时，file()和 linkFile() 函数是未知的，且名字自动是众所周知的。更重要的是，当如下调用 file() 时，得到的是保留的文件名：

```
QString fn = d.file();
```

上面这个函数产生一个文件名，并在硬盘上创建一个空的保留文件来阻止文件名被另一个应用程

序同时使用。在某些环境下，如果文件不存在，则不需要创建。如在 Qtopia 的文档 Tab 窗口中，图标代表 DocLnk 对象，DocLnk 对象仅仅为图标视图创建，它们没有相应的 .desktop 文件。为了避免产生垃圾保留文件，应查看这个文件是否有必要存在。

（3）功能

AppLnk 对象是通过对 .desktop 文件调用构造函数 AppLnk()来创建的，使用函数 isValid() 可以检查对象的有效性。

AppLnk 类的成员函数及其功能说明如表 5-13 所示。

表 5-13 AppLnk 类的成员函数及其功能说明

得到函数	设置函数	功能说明
name()	setName()	得到或设置应用程序的名字
pixmap()	无	得到应用程序的图标
bigPixmap()	无	得到应用程序的大图标
无	setIcon()	设置图标的文件名
type()	setType()	得到或设置文件的类型
rotation()	无	旋转 0、90、180 或 270 度
comment()	setComment()	得到或设置详细的评论信息
exec()	setExec()	得到或设置可执行的文件名
file()	无	得到文档的文件名
linkFile()	setLinkFile()	得到或设置.desktop 文件名
mimeTypes()	无	得到应用程序能查看或编辑的 MIME 类型
categories()	setCategories()	得到或设置功能描述
fileKnown()	无	判断文档的文件名是否存在
linkFileKnown()	无	判断.desktop 文件名是否存在
property()	setProperty()	得到或设置 AppLnk 属性

（3）DocLnkSet 和 DocLnk 类

DocLnkSet 类和 DocLnk 类的作用类似于 AppLnkSet 和 AppLnk 类，但其管理对象是针对文档的信息数据。

DocLnkSet 类从 AppLnkSet 类继承，用来管理文档连接 DocLnk 对象的集合，其构造函数创建了装有代表所有文档的 DocLnk 对象的容器类 DocLnkSet 的对象。

DocLnk 类从 AppLnk 类继承，它代表了装载的文档引用，其构造函数从一个有效的 .desktop 文件或一个新的.desktop 文件创建一个 DocLnk 对象。

示例：使用 DocLnk 类

下面这个例子使用 DocLnk 类来保存或装载文档文件：

```
void Main::setDocument( const QString& applnk_filename )
{
   FileManager fm;

   if ( current ) {
      if ( !fm.saveFile( *current, data ) ) {//把 data 存入到 current 文件
         // error
         return;
      }
      delete current;
```

```
    }

    current = new DocLnk( applnk_filename );

    if ( !fm.loadFile( *current, data ) ) {//从current文件中装载数据到data中
        // error
    }
}
```

产生文档的应用程序将总是通过添加一行说明到其.desktop文件中来声明支持的MIME类型，每个支持的类型将被定义在这个文件中，类型用“:”分开。所添加的一行说明列出如下：

```
MimeType=type/subtype;type/subtype;...
```

子类型可能是“*”，它表示应用程序能处理给定类型的所有形式，这一个应用程序将仅在没有其他可用的指定应用程序时才被触发。

在安装时，也可以选择运行类型：

```
qtopia-addmimetype type/subtype extension
```

（4）DocPropertiesWidget 和 DocPropertiesDialog 类

DocPropertiesWidget 类提供了对 AppLnk 属性更改的控制，它允许修改名字、位置和与特定 AppLnk 相连的文件的种类，这个文件能从 DocPropertiesWidget 中被删除、拷贝和传送。

DocPropertiesWidget 类从 QWidget 类继承，其构造函数创建了窗口部件来显示或操作 AppLnk 属性。DocPropertiesWidget 类的窗口部件如图 5-5 所示。

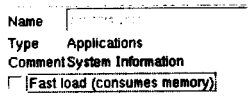


图 5-5 DocPropertiesWidget 类的窗口部件

DocPropertiesDialog 类允许用户检查与 AppLnk 对象相关的属性，它创建 DocPropertiesWidget 对象允许用户检查和修改与一个文件相关的属性。

（5）FileManager 类

FileManager 类辅助 AppLnk 的文档的输入/输出，它使用 loadFile()函数装载文档，使用函数 saveFile()存储文档，使用函数 copyFile()拷贝文档，使用函数 exists()测试文档是否存在，提供函数 openFile()来打开文档，使 I/O 设备准备读取。FileManager 类的成员函数及其功能说明如表 5-14 所示。

查看特定类型文件的应用程序称为“文档启动应用程序”，Qtopia 有框架支持以便简化应用程序。文档启动应用程序的顶层窗口部件的类声明中必须有一个 slot 槽函数的声明，列出如下：

```
public slot:
    void setDocument( const QString& applnk_filename );
```

表 5-14　FileManager 类的成员函数及其功能说明

成员函数名	功能说明
copyFile (const AppLnk & src, const AppLnk & dest)	拷贝文档 src 到文档 dest，如果操作成功，返回 TRUE；否则返回 FALSE
ensurePathExists (const QString & fn) [protected]	通过创建需要的目录来确保路径 fn 存在，如果操作成功，返回 TRUE；否则返回 FALSE
exists (const DocLnk & f)	如果文档 f 作为一个文件存在于硬盘上，返回 TRUE；否则返回 FALSE
loadFile (const DocLnk & f, QByteArray & ba)	从文档 f 装载文档到 ba，如果操作成功，返回 TRUE；否则返回 FALSE
loadFile (const DocLnk & f, QString & text)	这是为使用方便可重载的函数，它从文档 f 装载数据到字符串 text 中，text 必须是 UTF8 格式。如果操作成功，返回 TRUE；否则返回 FALSE
openFile (const DocLnk & f)	打开文档 f 作为一个可读的 QIODevice，如果操作成功，返回 QIODevice 对象；否则返回 0
saveFile (const DocLnk & f, const QString & text)	在文档 f 中存储 text 字符串，存储格式为 UTF8。如果操作成功，返回 TRUE；否则返回 FALSE
saveFile (const DocLnk & f, const QByteArray & data)	这是一个为方便使用的重载函数，在文档 f 中存储 data。如果操作成功返回 TRUE；否则返回 FALSE
saveFile (const DocLnk & f)	这是一个为方便使用的重载函数，它打开 f 作为一个可写的 QIODevice 设备并返回可写的 QIODevice 对象。如果操作失败，返回 0

这个 slot 函数应该用来存储应用程序的文档并显示或编辑指定的文档，一个实现的样例如下：

```
void Main::setDocument( const QString& applnk_filename )
{
    FileManager fm;

    if ( current ) {//如果当前文件连接存在
       if ( !fm.saveFile( *current, data ) ) {
          // error
          return;
       }
       delete current;
    }

    current = new DocLnk( applnk_filename );  //创建文档连接

    if ( !fm.loadFile( *current, data ) ) {//装载文档
       // error
    }
}
```

文档启动应用程序将总是通过添加一行到其 .desktop 文件中来声明支持的 MIME 类型，方式如下：

```
MimeType=type/subtype;type/subtype;…
```

subtype 可以是“ *”，表示这个应用程序能处理给定类型的所有格式。如果没有其他可用的应用程序，则这个应用程序被启动。

在安装时，也可以如下选择运行：

```
qtopia-addmimetype type/subtype extension
```

它允许普通的文件名（如插入设备的卡上的文件名）被识别作为 MIME 类型的文档。

5.8.3 PIM 管理类

Qtopia 的个人信息管理（PIM）库提供了一套安全的 API 访问 PIM 数据，可使用一些类来得到、编辑和存储地址簿的 Contact（联系人）和 Todo 任务，并且使应用程序与外部其他进程的变化保持更新。个人信息管理库相关类及其功能说明如表 5-15 所示。源代码在 src/libraries/qtopiapim 目录下。

表 5-15 个人信息管理（PIM）库相关类及其功能说明

类	功能说明
AddressBookAccess	安全访问地址簿数据的 API
AddressBookIterator	AddressBookAccess 类的迭代子 Iterator
DateBookAccess	访问 PimEvent 的 API
DateBookIterator	DateBookAccess 类的迭代子 Iterator
PimContact	持有地址簿条目的数据
PimEvent	持有一个日历事件的数据
PimRecord	在 Qtopia 数据库里 PIM 数据记录的基类
PimTask	持有 todo 条目的数据
TodoAccess	访问 PimTask 的 API
TodoIterator	TodoAccess 类的迭代子

下面对个人信息管理相关的类分别进行说明。

（1）PimRecord 类

PimRecord 类是 Qtopia 数据库中 PIM 数据记录的基类，它包含 PIM 应用程序使用的所有数据，用惟一的 ID 标识并且使用了一套分类。一个 PimRecord 的 ID 通过函数 uid() 返回，并使用函数 setUid() 设置。每个分类用一个整数代表，函数 categories() 返回 PimRecord 的分类，函数 setCategories() 设置 PimRecord 的分类。

（2）PimContact 类

PimContact 类持有地址簿条目的数据，其中包括个人名字信息、联系方式（contact）信息和诸如部门及工作头衔的商务信息。

（3）PimEvent 类

PimEvent 类从 PimRecord 类继承，它持有日历事件的数据，其中包括事件的描述数据和计划安排信息。

（4）PimTask 类

PimTask 类从 PimRecord 类继承，它持有 todo 条目的数据，其中包括任务的优先级、任务描述、可选的到期日期和任务是否完成标识。

（5）AddressBookAccess 和 AddressBookIterator 类

AddressBookAccess 类从 QObject 类继承，它提供访问地址簿数据的 API、尝试与修改保持同步，并在数据已被修改时提醒用户。AddressBookAccess 类的成员函数及其功能说明如表 5-16 所示。

表 5-16 AddressBookAccess 类的成员函数及其功能说明

成员函数名	功能说明
addContact (const PimContact & contact)	指定一个新的惟一 ID 给 contact，并添加 contact 到 pin 数据库中。增加 contact 要求 Contact 服务可用，如果 contact 正确添加，一个 addedContact(int,PimContact) 消息将被发送到 QPE/PIM 通道，其中 contact 能被用来决定指定的 ID

续表

成员函数名	功能说明
addressBookUpdated() [signal]	当地址簿中一个 contact 被增加、删除或修改时，这个信号被发送
contactForId (const QUuid & u, bool * ok = 0)	如果带有 u 的 contact 存在，返回 contact；否则返回 NULL。如果 ok 非空，在 contact 被发现时，设置 ok 为 TRUE；否则设置为 FALSE
editSupported ()	如果 pim 数据库中的 contact 能被增加、删除或更新，返回 TRUE；否则返回 FALSE
personalDetails ()	返回标识为 Personal Detail 的 PimContact
:removeContact (const PimContact & contact)	删除 pim 数据库的 contact
updateContact (const PimContact & contact)	更新 pim 数据库的 contact

AddressBookIterator 类为 AddressBookAccess 类提供了迭代子，仅仅通过它即可遍历一个 AddressBookAccess 的数据。AddressBookIterator 类的成员函数及其功能说明如表 5-17 所示。

表5-17 AddressBookIterator 类的成员函数及其功能说明

成员函数名	功能说明
AddressBookIterator (const AddressBookAccess & aba)	创建一个指向 aba 的开始处的 AddressBookIterator 对象
atFirst ()	如果迭代子在数据的第一个条目，返回 TRUE；否则返回 FALSE
atLast ()	如果迭代子在数据的最后一个条目，返回 TRUE；否则返回 FALSE
current ()	如果迭代子在一个有效的 PimContact 上，返回这个 PimContact 的指针；否则返回 0
operator* ()	操作符重载，如果迭代子在一个有效的 PimContact 上，返回这个 PimContact 的指针；否则返回 0
operator++ ()	操作符重载，设置迭代子到下一个条目。如果迭代子移到链表末尾外，返回 0；否则返回当前的 PimContact 的指针
operator= (const AddressBookIterator & other)	操作符重载，设置迭代子为 other 的拷贝
toFirst ()	设置迭代子到数据的第一个条目，如果 PimContact 存在，返回它；否则返回 0
toLast ()	设置迭代子到数据的最后一个条目。如果 PimContact 存在，返回它；否则返回 0

（6）DateBookAccess类

DateBookAccess 类提供了访问 PimEvent 事件的 API。PimEvent 事件被 Datebook 应用程序保存并在其他应用程序修改数据时，用来表示提供库的用户有数据被修改。 DateBookAccess 类的成员函数及其功能说明如表 5-18。

表 5-18 DateBookAccess 类的成员函数及其功能说明

成员函数名	功能说明
addEvent (const PimEvent & ev)	指定一个新的惟一 ID 给 ev，并添加事件到 pim 数据中。如果事件正确添加，一个 addedEvent(int,PimEvent)消息将被送到 QPE/PIM 信道上，这个消息中的事件将被用来判定指定的 ID
addException (const PimEvent & event, const QDate & date, const PimEvent & exception)	添加一个 exception 到 event 的 repeat 部分，event 将被阻止在给定的 date 日期发生，而是如 exception 的描述那样发生。exception 发生的日期不必与给定的 date 一样
addException (const PimEvent & event, const QDate & date)	添加一个 exception 到 event 的 repeat 部分，event 将被阻止在给定的 date 日期发生

续表

成员函数名	功能说明
dateBookUpdated () [signal]	当 datebook 被更新时发送的信号
editSupported ()	如果可能在 pim 数据中增加、删除和更新事件，返回 TRUE；否则返回 FALSE
eventForId (const QUuid & u, bool * ok = 0)	如果存在一个 ID 为 u 的事件，返回 event；否则返回 NULL。如果 ok 非空，在找到事件时，ok 被设为 TRUE；否则设为 FALSE
removeAllExceptions (const PimEvent & event)	为 event 的 repeat 模式删除所有的 exception，exception 将被正常的事件发生替代
removeEvent (const PimEvent & ev)	从 pim 数据中删除 ev
removeException (const PimEvent & event, const PimEvent & exception)	为 event 的 repeat 模式删除 exception，不为确认的 exception 比较数据。而是匹配 uid，并判定它是否匹配一个 event(它描述一个 repeat 模式的 exception)。这个 exception event 将被一个正常模式 event 的发生所替代。如果没有匹配的 exception，这个函数将无效
removeException (const PimEvent & event, const QDate & date)	删除 repeat 模式的 event 给定的 exception，这个 exception event 将被一个正常模式 event 的发生所替代。如果没有匹配的 exception，这个函数将无效
updateEvent (const PimEvent & event)	更新 pim 数据中的 event

（7）TodoAccess 类

TodoAccess 类提供了访问被 Todo 应用程序保存的 PimTask 的 API，尝试允许 Todo 应用程序对数据修改进行同步。并在其他应用程序做了修改时，告诉库的使用者。 TodoAccess 类的成员函数及其功能说明如表 5-19 所示。

表 5-19 TodoAccess 类的成员函数及其功能说明

成员函数名	功能说明
addTask (const PimTask & task)	指定一个新的惟一的 ID 给 task，并添加 task 到 pim 数据中。如果正确添加，一个 addedTask(int,PimTask)消息将被发送到 QPE/PIM 信道上。在这个消息中，task 被用来判定 ID
editSupported ()	如果可以增加、删除或更改 pim 数据中的任务（task），返回 TRUE；否则返回 FALSE
:removeTask (const PimTask & task)	从 pim 数据中删除所有的 task
taskForId (const QUuid & u, bool * ok = 0)	如果带有 u 的任务存，返回这个任务；否则返回 NULL。如果 ok 非空，任务被发现，ok 设置为 TRUE；否则设置为 FALSE
:todoListUpdated () [signal]	当 todl 链表被更新时，发出这个信号
updateTask (const PimTask & task)	更新在 pim 数据中的 task

5.8.4 桌面相关类

对基于设备的 Qtopia 来说，Qtopia 桌面是基于 PC 机的配套应用程序，用于为基于手机./PDA 上应用程序提供了鼠标和键盘便利。它提供了简单而直接的方式修改应用程序设置、同步数据和管理软件。

Qtopia 桌面提供如下功能。

（1）便于基于键盘输入的信息进入 Qtopia。

（2）查看 Qtopia 信息。

（3）在 Qtopia 桌面与移动设备间同步信息。

（4）传输文档。

（5）从 internet 网上或你的 PC 机上安装应用程序到嵌入设备。

（6）在第三方应用程序和 Qtopia 之间传输数据。

很容易扩展 Qtopia 桌面的功能。与 Qtopia 桌面相关的类有 Qtopia 桌面类、编译 Qtopia 桌面、Qtopia

桌面插件和 Qtopia 桌面同步等。

（1）Categories 和 CategoryGroup 类

Categories 和 CategoryGroup 类是同步 Qtopia 数据并与 Qtopia 桌面集成在一起的框架，一起组成了一个数据表格（或数据库），说明如下。

- Categories ：种类组数据库并映射 id 到名字，相当于一个数据表格。
- CategoryGroup：种类组，相当于一个数据表格的一栏，由多个条目构成。这里的种类表示一个应用程序类型，包含多个属性参数。

Categories 类是一个集合种类并映射 id 到名字的数据库，是底层种类访问的类。如果需要加一个菜单为应用程序过滤种类，可以使用 CategoryMenu 类。

Categories 类允许开发者增加、删除和重命名种类，可为诸如 Todo List 的单个应用程序而创建，或被所有的应用程序使用，被所有应用程序使用的 Categories 称为“全局种类”。每个 PimRecord 子类使用函数 PimRecord::setCategories() 和 PimRecord::categories()把种类作为一个 Qarray 保存，从而允许每个记录被指定到多个种类。也允许用户重命名一个种类，并为它自动更新所有的记录。

这个类提供了几个方法在种类 id 和与其相关的字符（如 id()、ids()、label()和 labels()）。一个名为“CategoryGroup”的辅助类被用来访问单个应用程序组（如 Todo List ）的种类，全局的种类也能通过 CategoryGroup 对象被访问。

Categories 保存在设置目录中的一个 xml 文件（Categories.xml ）中，一个名为“categoryFileName()”的函数将返回 Categories 所在的文件路径。

Categories 类使用 CategoryGroup mGlobalCats 来保存全局种类，使用 QMap< QString, CategoryGroup > mAppCats 来保存各个应用程序的种类。

CategoryGroup 类是与一个应用程序相关或全局的一组种类，它定义了由 id 到种类标签和由种类标签到 id 映射，可以使用标签或惟一的 id 进行查找。这是 Categories 类使用的内部类，该类在 Categories::appGroupMap()和 Categories::globalGroup()中返回这个类对象。CategoryGroup 类的成员函数及其功能说明如表 5-20 所示。

表 5-20 CategoryGroup 类的成员函数及其功能说明

成员函数名	功能说明
contains (int uid)	如果 uid 存在种类组中，返回 TRUE；否则返回 FALSE
contains (const QString & label)	如果标签存在种类组中，返回 TRUE；否则返回 FALSE
:id (const QString & label)	返回与 label 相连的 uid，如果未找到，返回 0
idMap ()	返回 id 到标签的 QMap 映射链表的引用
ids ()	为方便使用提供的重载函数，它返回与种类组相关的 id 的链表
label (int uid)	返回与 uid 相连的标签字符串，如果未发现，返回 NULL
labels ()	返回存在种类组中的所有标签的字符串链表
labels (const QArray<int> & catids)	返回与 catids 相关的所有标签的字符串链表

CategoryGroup 类使用下面的类保存储 id 和标签：

```
QMap<int, QString> mIdLabelMap;  //id    到标签的映射
QMap<QString, int> mLabelIdMap;  //标签到 id 的映射
```

（2）CategoryMenu 和 CategorySelect 类

CategoryMenu 类是一个窗口部件，它从 QPopupMenu 类继承，被 Category 用来辅助过滤记录或文件。它提供一个弹出菜单，使得 category 更容易过滤记录或文件。它查找一个应用程序的可用种类后放到菜单上，并保持跟踪过滤的种类。 CategoryMenu 类的成员函数及其功能说明如表 5-21 所示。

表 5-21 CategoryMenu 类的成员函数及其功能说明

成员函数名	功能说明
CategoryMenu (const QString & application, bool globals, QWidget * parent = 0, const char * name = 0)	构造函数，创建名为 name 的新 CategoryMenu 对象，菜单将得到 application 的可用的种类。如果 global 为 TRUE，那么还将得到全局种类
categoryChange () [signal]	当用户在菜单上选择不同的种类时，发出这个信号通知所选的记录或文件已改变
isSelected (const QArray<int> & cUids)	如果带有一系列种类 ID 为 cUids 的记录或文件被当前选择选中，返回 TRUE；否则返回 FALSE
reload () [slot]	重得到可用的种类，传给窗口部件
setCurrentCategory (int newCatUid)	设置菜单，把 newCatUid 作为当前被选中的种类
setCurrentCategoryAll ()	设置菜单允许所有的种类被设置
setCurrentCategoryUnfiled ()	设置菜单仅允许空的种类被设置

CategorySelect 窗口部件允许用户使用 combobox 界面选择种类。它提供了种类的 QComboBox 控件，用来过滤（如在联系人表视图中）或允许用户选择多个种类。如果种类被改变， CategorySelect 自动更新自身。信号与槽被用来通知用户选择所在的应用程序，一个 QToolButton 被提供以便用户能手动编辑种类。

5.8.5 字体与输入法相关类

字体与输入法相关类说明如下。

（1）FontDatabase 和 FontFactoryInterface 类

FontDatabase 类从 QFontDatabase 继承，它提供了可用字体的相关信息并使用 families()函数查询可用的库。当需要访问不可用的字体时，应使用 FontDatabase ，而不是 QFontDatabase 类。例如：如果 freetype 类型库和 Qtopia freetype 类型插件被安装，则应用程序可使用 TrueType 字体。字体产生者插件比系统字体需要更大的资源，因此字体产生者插件仅在需要时才使用，可以使用 loadRenderers() 装载字体产生者插件。

FontFactoryInterface 类为添加字体产生器到 Qtopia 中提供了接口，通过插件字体工厂可加入到 Qtopia 中。为了写一个字体工厂插件，必须通过从 FontFactoryInterface 类派生类并运用函数 fontFactory() 创建到 QFontFactory 派生类的接口，并且必须从 QRenderedFont 派生类来提供实际的字体产生方法。

（2）InputMethodInterface 和 ExtInputMethodInterface 类

InputMethodInterface 类提供了 Qtopia 弹出式输入方法的接口，该输入法提供了一个窗口部件显示在屏幕上并在键被按下时发出一个信号。弹出输入法可通过插件加入到 Qtopia 中，该类提供了弹出输入法接口的需要的虚拟函数，由开发者从这个类的继承类来实现这些函数。

ExtInputMethodInterface 类提供了输入法的扩展接口，可用于弹出输入法和复合输入法。

复合输入法从一个独立的键盘（实际或虚拟的）修改输入，在这种情况下使用函数 inputMethod() 和 statusWidget()。复合输入法主要用于中文、日文和韩文的输入，但也能用于任何文本输入的能力扩展，包括从弹出输入法进行扩展。在特定的弹出输入法和复合输入法之间可以有依赖关系，这个依赖关系可能通过函数 compatible() 来定义。

输入法可通过插件加入到 Qtopia 中，可以从这个类继承来创建自己的输入法接口，实现 ExtInputMethodInterface 类的虚函数。

5.8.6 StorageInfo 和 FileSystem 类

在 Qtopia 中，整个文件系统的信息用 StorageInfo 类描述，每个挂接点对应的文件系统信息使用 FileSystem 类描述。在 StorageInfo 类中有一个 FileSystem 对象链表 mFileSystems 用于保存所有存在的文件系统信息。

（1）StorageInfo 类

StorageInfo 类描述了挂接在文件系统上的硬盘情况，这个类提供了对文件系统的挂接信息的访问，每个 FileSystem 类对象代表一个文件系统挂接点。为了确保得到硬盘最新的大小信息，需要调用 update() 方法。当硬盘挂接/卸载时，它将被操作系统的信号自动调用。

update()方法从 /etc/mtab 文件中读取文件系统挂接点信息，创建 FileSystem 类对象放入到 FileSystem 类对象链表中。StorageInfo 类的成员函数及其功能说明如表 5-22 所示。

表 5-22 StorageInfo 类的成员函数及其功能说明

StorageInfo 类成员函数	功能说明
disksChanged () [signal]	当一个硬盘挂接或卸载时发出的信号，如一个 CF 卡已插入
fileSystemNames ()	返回文件系统名的链表
fileSystemOf (const QString & filename)	返回匹配挂接点名字的文件系统
fileSystems ()	返回所有可用的挂接文件系统的链表
update () [slot]	更新每个挂接点的挂接信息和自由空间信息

StorageInfo 类的成员函数 update 用来更新文件系统的信息，它使用文件系统描述文件及其操作函数。下面对函数 update 进行分析，分析函数之前，首先说明标准 C 库的文件系统描述文件及操作函数。

在 Linux 中，各种文件系统的信息保存在文件/etc/fstab 和/etc/mtab 中。其中/etc/fstab 是文件系统描述文件，/etc/mtab 是已挂接的文件系统描述文件。两个文件中的每一行的域可用下面的结构描述：

```
struct mntent {
    char   *mnt_fsname;       /* 已挂接文件系统名 */
        char   *mnt_dir;      /* 文件系统路径前缀 */
        char   *mnt_type;     /* 挂接的文件系统类型 */
        char   *mnt_opts;     /* 挂接选项 */
        int    mnt_freq;      /* dump 频率（天数） */
        int    mnt_passno;    /* pass number on parallel fsck*/
};
```

在标准 C 库中提供了对文件/etc/fstab 和/etc/mtab 的操作函数，这些函数列出如下：

```
#include <stdio.h>
   #include <mntent.h>
   FILE *setmntent(const char *filename, const char *type);
   struct mntent *getmntent(FILE *filep);
   int addmntent(FILE *filep, const struct mntent *mnt);
   int endmntent(FILE *filep);
   char *hasmntopt(const struct mntent *mnt, const char *opt);
```

函数 setmntent() 打开文件系统描述文件 filename 得到 FILE 文件指针，函数 getmntent()通过文件指针 filep 读取文件的每一行，每一行得到一个结构 mntent 的数据。这样，函数 getmntent()返回指针多个结构 mntent 的指针，函数 addmntent()添加一个结构 mntent 数据到文件 filep 的结尾，函数 endmntent() 关闭打开的文件 filp，函数 hasmntopt()扫描参数 mnt 的 mnt_opts 域，查看是否有一个子字符串匹配 opt。

函数 update 从配置文件 Storage 或从挂接的文件系统描述文件/etc/mtab 中读出文件系统的信息，以使得 FileSystem 对象保存到对象链表 mFileSystems 中，更新文件系统存储容量信息。函数 update 分析如下：

```
void StorageInfo::update()
{
    //对于 Linux操作系统
#if defined(_OS_LINUX_) || defined(Q_OS_LINUX)
    struct mntent *me;
    FILE *mntfp = setmntent( "/etc/mtab", "r" ); //    打开文件

    Config cfg("Storage"); // 从配置文件 Storge得到 Config对象

    QStringList curdisks;
    QStringList curopts;
    QStringList curfs;
    bool rebuild = FALSE;
    int n=0;
    if ( mntfp ) {
        //遍历/etc/mtab 每一行，得到每一行的 mntent 结构实例 me
        while ( (me = getmntent( mntfp )) != 0 ) {
            QString fs = me->mnt_fsname;
            bool relevant = FALSE;

            if ( cfg.isValid() ) {// 配置文件 Storage存在
                cfg.setGroup(fs);
                relevant = !cfg.readEntry("Name").isEmpty();
            } else {
                relevant = fs.left(7)=="/dev/hd" || fs.left(7)=="/dev/sd"
                    || fs.left(8) == "/dev/ram"
                    || fs.left(8) == "/dev/mtd"
                    || fs.left(9) == "/dev/mmcd"
                    || fs.left(14) == "/dev/mmc/part1"
                    || fs.left(9) == "/dev/root"
                    //故意不考虑 tmpfs，因为它不是文件的安装位置，它只是/tmp 的映射
                    // || fs.left(5)=="tmpfs"
                        // "which-qtopia" may be running off the SD card
                        && (QString(me->mnt_dir)!="/home/QtPalmtop"
                            && QString(me->mnt_dir)!="/opt/Qtopia") ;
            }

            if ( relevant ) {//如果文件系统存在
                n++;
                curdisks.append(fs); //将挂接的文件系统名存入字符串链表
                curopts.append( me->mnt_opts ); //将挂接时的命令选项存入字符串链表
                //qDebug("-->fs %s opts %s", fs.latin1(), me->mnt_opts )       ;
                curfs.append( me->mnt_dir ); //将挂接的目录存入字符串链表
                bool found = FALSE;
                for (QListIterator<FileSystem> i(mFileSystems) ; i.current(); ++i) {
                    if ( (*i)->disk() == fs ) {// 如果 FileSystem 对象链表中存在 fs
                        found = TRUE;
                        break;
                    }
                }
                if ( !found )// 如果 FileSystem对象链表中不存在 fs，需要重构建
                    rebuild = TRUE;
            }
        }
        endmntent( mntfp ); // 关闭文件
    }

    //重构建 FileSystem 对象链表 mFileSystems
    if ( rebuild || n == 0 || n != (int)mFileSystems.count    () ) {
        mFileSystems.clear(); //清除链表
        QStringList::ConstIterator it=curdisks.begin();
        QStringList::ConstIterator fsit=curfs.begin();
        QStringList::ConstIterator optsIt=curopts.begin();
        for (; it!=curdisks.end(); ++it, ++fsit, ++optsIt) {
            QString opts = *optsIt;
```

```
            QString humanname=*it;
            bool removable = FALSE;
            if ( cfg.isValid() ) {//配置文件 Storage 存在
                cfg.setGroup(*it);
                humanname = cfg.readEntry("Name",humanname);
                removable = cfg.readBoolEntry("Removable",FALSE);
            } else if ( isCF(humanname) ) {
                humanname = tr("CF Card");
                removable = TRUE;
            } else if ( humanname == "/dev/hda1" ) {
                humanname = tr("Hard Disk");
            } else if ( humanname.left(9) == "/dev/mmcd" ) {
                humanname = tr("SD Card");
                removable = TRUE;
            } else if ( humanname.left(7) == "/dev/hd" )
                humanname = tr("Hard Disk") + " " + humanname.mid(7);
            else if ( humanname.left(7) == "/dev/sd" )
                humanname = tr("SCSI Hard Disk") + " " + humanname.mid(7);
            else if ( humanname == "/dev/mtdblock1" || humanname == "/dev/mtdblock/1" )
                humanname = tr("Internal Storage");
            else if ( humanname.left(14) == "/dev/mtdblock/" )
                humanname = tr("Internal Storage") + " " + humanname.mid(14);
            else if ( humanname.left(13) == "/dev/mtdblock" )
                humanname = tr("Internal Storage") + " " + humanname.mid(13);
            else if ( humanname.left(9) == "/dev/root" )
                humanname = tr("Internal Storage") + " " + humanname.mid(9);
            else if ( humanname.left(8) == "/dev/ram" )
                humanname = tr("RAM disk") + " " + humanname.mid(8);
            FileSystem *fs = new FileSystem( *it, *fsit, humanname, removable, opts );
            mFileSystems.append( fs );  //将文件系统信息存入对象链表 mFileSystems
        }
    #endif
            //检查 mFileSystems 是否有 FileSystem 对象存在
            if (mFileSystems.count() == 0) {
               FileSystem *fs = new FileSystem( "/dev/hda1",
                   QDir::homeDirPath(), tr("Internal Storage"), FALSE, "");
               mFileSystems.append( fs );
            }
            emit disksChanged();//发信号
    #if defined(_OS_LINUX_) || defined(Q_OS_LINUX)
       } else {
            //仅更新存储容量信息
            for (QListIterator<FileSystem> i(mFileSystems); i.current(); ++i)
                i.current()->update();
       }
    #endif
    }
```

（2）FileSystem类

FileSystem 类描述了单个挂接点，并返回挂接点的信息，包括文件系统名、挂接点、可读的名字、大小信息和挂接点选项等。FileSystem 类的成员函数及其功能说明如表 5-23 所示。

表 5-23 FileSystem 类的成员函数及其功能说明

类成员函数名	功能说明
availBlocks ()	返回文件系统可用的块数
blockSize ()	返回文件系统每块的大小
disk ()	返回文件系统的名，如/dev/hda3
isRemovable ()	是否可插拔的卡
isWritable ()	文件系统是否作为可写挂接的，可读写，返回 TRUE；只读，返回 FALSE
name ()	返回挂接可被读取的目录的名字
options ()	返回挂接的选项

续表

类成员函数名	功能说明
path ()	返回挂接的路径，如/home
totalBlocks ()	返回文件系统总的块数
update()	更新文件系统存储容量信息

5.8.7 DeviceButtonManager 和 DeviceButton 类

DeviceButtonManager 类管理设备按钮映射，在一个设备上的功能按钮可提供按键和激活键的服务。该类允许功能按钮的角色被访问或被修改。 DeviceButtonManager 类的成员函数及其功能说明如表 5-24 所示。

表 5-24 DeviceButtonManager 类的成员函数及其功能说明

成员函数名	功能说明
buttonForKeycode (ushort keyCode)	返回 keyCode 的 DeviceButton 对象，如果 keyCode 未被发现，返回 0L
buttons ()	返回设备上可能的按钮，按钮的序号和位置将依赖于设备而改变。按钮的序号将由设备生产商指定，从首选的按钮到最少的按钮排列。注意这个链表仅包含“用户映射”的按钮
factoryResetButtons ()	重设置按钮回到工厂的默认设置
instance () [static]	返回 DeviceButtonManager 的静态实例
remapHeldAction (int button_index, const ServiceRequest & action)	重指定 action 的 held（保持）动作给 button_index 按钮
remapPressedAction (int button_index, const ServiceRequest & action)	重指定 action 的 press（按）动作给 button_index 按钮

DeviceButton 类代表在一个 Qtopia 设备上物理的用户映射按钮，一个设备可能有“用户编程”的按钮，按钮的位置和序号将根据设备不同而改变。 DeviceButton 类的成员函数及其功能说明如表 5-25 所示。

表 5-25 DeviceButton 类的成员函数及其功能说明

成员函数名	功能说明
heldAction ()	当这个按钮被按住时返回用户指定的动作
keycode ()	返回按钮的键代码
operator== (const DeviceButton & e)	如果两个 DeviceButton 对象相等，返回 TRUE。 pixmap () 返回这个按钮的位图，如果没有，返回空
pixmap ()	返回这个按钮的位图，如果没有，返回空
pressedAction ()	当按钮被按时，返回用户指定的动作
setHeldAction (const ServiceRequest & action)	设置按钮被按住时的动作
setKeycode (ushort keycode)	设置按钮的键代码
setPixmap (const QString & pmn)	设置按钮的位图
setPressedAction (const ServiceRequest & action)	设置按钮被按下时的动作
setUserText (const QString & text)	设置可读且翻译的按钮描述
userText ()	返回按钮的可读且翻译的描述

5.8.8 图像处理相关类说明

图像处理相关类说明如下。

（1）ImageCodecInterface 类

ImageCodecInterface 类允许可插拔的 image 格式，它提供了函数 installIOHandler 安装 Qt image IO 处理器，函数 keys 返回 codec 支持的格式的链表。

（2）ImageSelector 类

ImageSelector 类窗口部件允许用户从一个图像集合中选择图像，预览一个图像集合并从集合中选择单个图像。

可以通过函数 setViewMode()控制图像显示的方式，ImageSelector 支持两种显示方式，一种为 ImageSelector::Single，图像可以放大或缩小以适合 ImageSelector 窗口部件的大小，这种方式允许用户依次巡视图像；另一种方式是 ImageSelector::Thumbnail，图像可以放大或缩小并以群组形式显示（类似于缩略图显示方式），函数 setThumbnailSize()能控制图像放大的最大尺寸。当用户已选择图像时，ImageSelector 类发出 selected()信号。选中图像的 DocLnk 对象将传递这个信号，可通过函数 selectedFilename()或 selectedImage()返回选择的图像。函数 selectedImage() 返回一个装有选择图像的拷贝的 QPixmap 对象。

（3）ImageSelectorDialog 类

ImageSelectorDialog 窗口部件从 ImageSelector 类继承，它是一个对话框式的 ImageSelector 窗口部件，允许用户从一个图像集中选择图像。

示例：选择墙纸

下面的代码允许用户从 Qtopia 目录中选择一种墙纸：

```
QStringList locations;
QStringList qtopiaPaths = Global::qtopiaPaths();
for( QStringList::Iterator it = qtopiaPaths.begin(); it != qtopiaPaths.end(); ++it )
    locations.append( *it + "pics/wallpaper" );

ImageSelectorDialog dialog( locations, this );
if( QPEApplication::execDialog( &dialog ) ) {
   // 接受
} else {
   // 拒绝
}
```

可以使用函数 selectedDocument()、selectedFilename()和 selectedImage() 返回选择的图像，函数 selectedImage()返回装有选中图像拷贝的 QPixmap 对象。

5.8.9 多媒体播放相关类说明

多媒体播放相关类说明如下。

（1）LibMadPlugin 类

LibMadPlugin 类是库 libmad 的包裹器，LibMAD 是基于整数的 MPEG 音频解码器，支持所有的 3 层 MPEG 层并被媒体播放器使用。它根据 GNU General Public License 发布。

（2）FrameBuffer 类

FrameBuffer 类是媒体播放器应用程序特定的类，在内存中申请一块建立成帧缓冲格式，还可使用 FrameBuffer 类的 rotateToBuffer 函数对帧缓冲中的画面进行放置可使用 QImage 类或 QDirectPainter 类对帧缓冲进行直接绘制。这个类是 VideoOutput 使用的特定的类，允许解码后的视频帧直接输出到屏幕。

（3）FFMpeg 类

FFMpeg 类是 license 信息的占位符，avformat 和 avcodec 代码是从 FFMPEG 项目中提取，FFMPEG 是提供记录、变换音频和视频数据的规范的项目。 avformat 和 avcodec 被媒体播放器使用，FFMPEG 项目在 GNU Lesser General Public License 下发布。

（4）MediaPlayer 类

MediaPlayer 类是媒体播放器特定的类，创建播放媒体文件的所有要素，它使用 createMediaSelector()、createAudioUI() 和 createVideoUI() 虚函数来抽象 GUI 元素。这样可能对同样的后台生成不同的前端，使用同样的 API函数。

（5）MediaPlayerDecoder 类

MediaPlayerDecoder 类为 Qtopia 媒体播放器解压缩插件提供一个抽象的基类，从这个类派生类来产生解压缩插件并重载这个类的虚函数，使用宏 Q_EXPORT_PLUGIN 来输出这个类。

（6）MediaPlayerPluginInterface 类

MediaPlayerPluginInterface 类用来访问 MediaPlayerDecoder 对象。

（7）MediaPlayerState 类

MediaPlayerState 类用来保存并通知媒体插放器应用程序的各种状态和选项的改变。

（8）MediaRecorderEncoder 类

MediaRecorderEncoder 类提供媒体录像大的编码插件的基类，从这个类派生类并重载虚函数即可实现一个编码插件，使用宏 Q_EXPORT_PLUGIN 输出插件。

在录音时，这个类的函数如下使用：

```
begin(device, tag);
addComment("name", "value");
...
addComment("name", "value");
setAudioChannels(2);
setAudioFrequency(44100);
writeAudioSamples(samples, numSamples);
...
writeAudioSamples(samples, numSamples);
end();
```

（9）MediaRecorderPluginInterface 类

MediaRecorderPluginInterface 类提供了一个获得多媒体编码的方法，它允许应用程序获得一个编码器来记录多媒体数据到一个文件或其他输出设备。

（10）MediaScanningProgress 类

MediaScanningProgress 类在搜索文档时显示一个进度条，表示搜索的进度。

（11）MediaSelector 类

MediaSelector 类列出 MediaSelectorItems，通过各种媒体特定的属性、id3 标记或其他规则或以列出媒体条目。

（12）MediaSelectorWidget 类

MediaSelectorWidget 类装载并显示 MediaSelector 或 MediaScanningProgress。

5.8.10 网络相关类说明

网络相关类说明如下。

（1）NetworkInterface 类

NetworkInterface 类允许插入网络服务，Qtopia 网络设备由目录 etc/network/下的配置文件定义。例如，一个通用的 DialUp.conf 配置文件如下（实际的文件用 UTF-8 编码）：

```
[Info]
Name = Dial-up - PPP
Type = dialup
Layer = TCP/IP
[Help]
Help = For dial-up connections through a modem and plain telephone line.
Help[de] = Für Wählverbindungen mit Modem und Telefon.
[Properties]
Auto = 0
Device = ppp
CardType = serial
```

为增加一个指定的拨号服务，仅需提供另一个配置文件，列出如下：

```
[Info]
Name = SuperNet ISP
Type = dialup
Layer = TCP/IP
[Help]
Help = Connect to your local SuperNet(tm) point-of-presence.
[Properties]
Auto = 0
Device = ppp
CardType = serial
password = secret
phone = 555555555
speed = 115200
username = customer
```

为了增加一个不在已有的服务范围的服务，可通过创建一个 NetworkInterface **插件定义自己的网络服务类型。**

NetworkInterface **类的成员函数及其功能说明如表** 5-26 **所示。**

表 5-26　NetworkInterface　类的成员函数及其功能说明

成员函数名	功能说明
addStateWidget (QWidget * parent, Config & cfg) [virtual]	返回一个窗口部件，parent 是其父窗口。它显示 cfg 定义的网络服务的状态，默认返回 0，表示没有状态窗口部件
cardType (Config & cfg) [virtual]	返回与这个服务相关的 pcmcia 卡类型，默认情况下在 cfg 中的 Properties.CardType 值被返回
create (Config & cfg) [pure virtual]	执行 cfb 定义的网络服务的系统状态切换，这个函数仅在服务第 1 次被用户加入时调用
device (Config & cfg) [virtual]	返回用于连接的网络设备，默认时 cfg 中的 Properties.Device 值被返回
doProperties (QWidget * parent, Config & cfg) [pure virtual]	弹出提示对话框，用户手动设置 cfg 定义的网络服务的属性
isActive (Config & cfg) [virtual]	如果 cfb 定义的服务被激活，返回 TRUE。默认时，在目录/proc/net/dev 下 device()被搜索
isAvailable (Config & cfg) [virtual]	如果 cfg 定义的网络服务是可以被激活的，返回 TRUE。默认时，在系统的 pcmcia 表中，cardType()被搜索
needPassword (Config & cfg) [virtual]	如果 cfg 定义的网络服务需要一个口令，返回 TRUE
remove (Config &) [pure virtual]	恢复在 create()中改变的系统状态
start (Config & cfg) [pure virtual]	开始 cfg 定义的网络服务，如果操作成功，返回 TRUE
start (Config & cfg, const QString & password) [virtual]	开始 cfg 定义的网络服务，使用给定的口令。如果操作成功，返回 TRUE。默认情况下调用没有口令的 start()方法
stop (Config & cfg) [pure virtual]	停止 cfg 定义的网络服务，操作成功，返回 TRUE

（2）GSM 类

GSM 类是 license 信息的占位符，GSM 是记录人类讲话的 codec，其代码来源于从 Sox 声音工具的 12.17.3 版本。

（3）Ir 类

Ir 类支持在一个红外通信连接中发送对象，DocLnk 对象和文件可通过红外连接发送。当发送完成时，发出 done()信号，函数 supported()用来判断设备是否支持红外通信。

（4）Konsole 类

Konsole 类是终端（embeddedkonsole）应用程序的顶层窗口部件，终端允许命令行访问系统，以及用 于开发和调试。终端是基于 KDE 的控制台 Konsole 应用程序，并在 GPL License 下发布。

（5）KVNC 类

KVNC 类是一个 VNC 客户端，Keypebble（Qtopia 的 VNC 客户端，基于 KDE 项目中的 Keystone、Keystone 和 Keypebble）在 GPL License 下发布，VNC 服务器代码版权属于 Trolltech 公司。

5.8.11 软件模块接口类

为了提供软件配置的灵活性，Qtopia 提供了各种接口类，这样用户可以灵活地加入自己的软件模块这些软件模块大多数以插件的形式，由插件装载器动态装入，还有少数惟一存在的软件模块在编译时被选择进入系统。Qtopia 接口类的组织结构如图 5-6 所示。

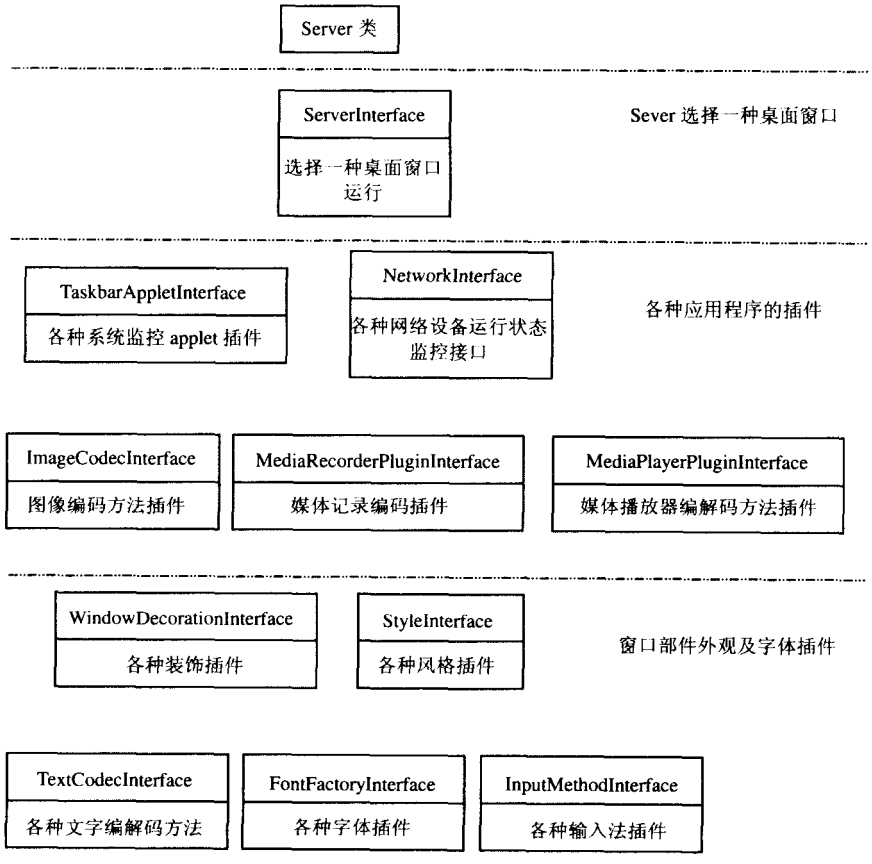


图 5-6 Qtopia接口类组织结构

下面是部分接口类的说明。

（1）WindowDecorationInterface 类

WindowDecorationInterface 类从 QUnknownInterface 类继承，为 Qtopia 窗口装饰风格提供了一个接口，窗口装饰风格可以通过插件加入到 Qtopia 中。为了写一个风格插件，必须通过从 WindowDecorationInterface 派生类来创建一个接口到 QStyle 派生类，并实现纯虚函数。

被装饰的窗口由结构 WindowDecorationInterface 用来定义窗口数据，这个结构列出如下：

```
struct WindowData {
    QRect rect;
    QPalette palette;
    QString caption;
    enum Flags { Maximized=0x01, Dialog=0x02, Active=0x04 };
    Q_UINT32 flags;
    Q_UINT32 reserved;
};
```

WindowDecorationInterface 类的成员函数及其功能说明如表 5-27 所示。

表 5-27 WindowDecorationInterface 类的成员函数及其功能说明

成员函数名	功能说明
drawArea (Area area, QPainter * p, const WindowData * wd)	使用 p 绘制窗口 wd 特定的区域 area
drawButton (Button b, QPainter * p, const WindowData * wd, int x, int y, int w, int h, QWSButton::State state)	使用 p 绘制窗口 wd 在 x, y, w, h 范围内状态为 state 的按钮
icon ()	返回装饰的图标位图，可显示在外观设置对话框中
mask (const WindowData * wd)	返回装饰的掩码，包括窗口 wd 所有的边界和标题。窗口装饰不一定是矩形的，但标题区域必须是矩形的并且必须是窗口的宽度，这样确保标题被正确绘画在最大化窗口中
metric (Metric m, const WindowData * wd)	返回窗口 wd 的 m 类型对应的值，如标题的高度
name ()	返回装饰的名字，将显示在外观设置对话框中

（2）StyleInterface 类

StyleInterface 类从 QUnknownInterface 继承，为 Qtopia 窗口部件风格提供了一个接口，风格可以通过插件加入到 Qtopia中。为了写一个风格插件，必须通过从 StyleInterface 派生类并实现其虚函数来创建一个接口到 QStyle派生类。其成员函数函数 name()返回风格的名字，它将显示在外观设置对话框中。函数 StyleInterface::style()返回风格类对象实例，如果这个函数被调用多次，则总是返回同样一个风格类对象实例。

（3）ServerInterface 类

ServerInterface 类为 Qtopia用户 laucher 提供了一个接口，允许用户优化 launcher 接口以更适合于设备。对于 PDA 风格设备，推荐使用默认的 laucher。专有设备可以选择提供更适合于应用程序的 launcher。对于任何特定的设备来说， launcher 是固定的，而不能作为插件装载。 launcher 必须通过 server.pro 和 server.cpp 编译为 server 应用程序。ServerInterface 类的成员函数及其功能说明如表 5-28 所示。

表 5-28 ServerInterface类的成员函数及其功能说明

成员函数名	功能说明
allApplicationsRemoved () [pure virtual]	从 launcher 中删除所有的应用程序
allDocumentsRemoved () [pure virtual]	从 launcher 中删除所有的文档
appLnks () [static]	返回可用应用程序的 AppLnkSet 类对象

续表

成员函数名	功能说明
applicationAdded (const QString & type, const AppLnk & app) [pure virtual]	添加一个 type 类型的应用程序 app 到 launcher 中
applicationRemoved (const QString & type, const AppLnk & app) [pure virtual]	从 launcher 中删除一个 type 类型的应用程序 app
applicationScanningProgress (int percent) [pure virtual]	系统正在扫描已安装的应用程序，percent 表示文件系统被扫描的百分比。当扫描开始时，这个函数被调用，percent 为 0。扫描完成时，为 100
applicationStateChanged (const QString & name, ApplicationState state) [pure virtual]	应用程序 name 已变化到状态 state，用来显示 launch 通知并更新正运行应用程序的链表
createGUI () [pure virtual]	创建用户的 luncher UI 界面
destroyGUI () [pure virtual]	销毁用户 launher UI 界面，所有被 createGUI()分配的资源将被释放
dockWidget (QWidget * w, DockArea placement) [static]	将顶层窗口部件 w 停靠在屏幕被 placement 定义的一侧，窗口部件按其停放的次序、sizeHint()和以前停靠的窗口部件是否可见来决定放置的位置。QWidget::showMaximized()将使整个桌面被这个窗口所用，例如，如果一个窗口部件停靠在屏幕的底部，则 sizeHint()将定义其高度，且将使用屏幕的整个宽度。如果一个窗口部件停靠在右侧，则 sizeHint()将定义其宽度，并且将在没有掩盖停靠在底部的窗口部件的情况下尽可能地高。这个函数对于保留系统区域是有用的，如保留任务栏和输入法区域，它们不应该被应用程序覆盖
documentAdded (const DocLnk & doc) [pure virtual]	添加一个文档 doc 到 launcher 中
documentChanged (const DocLnk & oldDoc, const DocLnk & newDoc) [pure virtual]	将一个文档的属性从 oldDoc 改变到 newDoc
documentRemoved (const DocLnk & doc) [pure virtual]	为 launcher 删除一个文档
documentScanningProgress (int percent) [pure virtual]	系统扫描文档
requiresApplications () const [pure virtual]	如果用户 launcher 请求服务器扫描应用程序，返回 TRUE
requiresDocuments () const [pure virtual]	如果用户请求服务器扫描文档，返回 TRUE
showGUI () [pure virtual]	实现这个函数第 1 次显示用户 launcher UI 界面，这个函数在 createGUI()之后被调用
storageChanged (const QList<FileSystem> &) [pure virtual]	一个文件系统已被增加或删除
typeAdded (const QString & type, const QString & name, const QPixmap & pixmap, const QPixmap & bgPixmap) [pure virtual]	加入 type 种类的应用程序，如 type 为 Game 种类。参数 name 是 category 的翻译名，type 能在任何时候被加入或删除，例如：当一个含有非标准的 category 的存储卡被插入或被拔去时
typeRemoved (const QString & type) [pure virtual]	一个应用程序 category 的 type 已被删除

5.8.12 Service 和 ServiceRequest 类

Service 和 ServiceRequest类说明如下。

（1）Service 类

Service 类允许应用程序为其他应用程序提供服务，一个服务是一个命名的特征集合，如网络浏览器提供了显示给定 URL 的网页的特征。服务允许一个服务的提供者（如一个网页浏览器）来自这个服务的消费者中，如一个希望用户查看 URL 的邮件读者。Service 类的成员函数及其功能说明如表 5-29 所示。

表 5-29 Service 类的成员函数及其功能说明

成员函数名	功能说明
serviceList_p(const QString& d, const QString n)	这个函数递归运行，将目录 d 下的文件名全部放在一个字符串链表中并返回这个链表，n 是添加的前缀
list()	调用函数 serviceList_p 将 services 目录下的所有文件名放在一个字符串链表中并返回这个链表
binding(const QString& service)	将 sevice 字符串中的 / 用 _ 代替，并加上 Service-前缀返回
config(const QString& service)	将 sevice 对应的全路径名返回
apps(const QString& service, bool automatics)	返回提供 service 的所有应用程序，如果无，返回 null。如果 automatics 为 TRUE（默认值），MimeType 数据库和可运行的应用程序也搜索服务提供者。可执行应用程序的搜索仅向后搜索
app (const QString & service, const QString & appname = QString::null, bool automatics = TRUE) [static]	返回提供 service 的应用程序，如果无，返回 null 字符串。如果提供了 appname，一个对这个应用程序的特定绑定被使用，而不是全局绑定。如果 automatics 是 TRUE（默认值），并且 service 未定义，MimeType 数据库和可执行的应用程序也被搜索 service 提供者。可执行的应用程序仅被向后搜索
appConfig (const QString & service, const QString & appname = QString::null, bool automatics = TRUE) [static]	返回提供 service 的应用程序的配置文件，配置文件提供了更详细的有关应用程序提供服务的特定信息，应用程序总是提供定义在配置文件中的最少的服务。如果参数 appname 存在，将使用这个应用程序的特定绑定
appLnk (const QString & service, const QString & appname = QString::null, bool automatics = TRUE) [static]	返回提供 service 应用程序的 AppLnk 对象。如果参数 appname 存在，将使用到这个应用程序的特定绑定
apps (const QString & service, bool automatics = TRUE) [static]	返回提供 service 服务的所有应用程序
binding (const QString & service) [static]	返回定义有用户给 service 选择的绑定的配置文件名字，它在 service 加上前缀 serice-
channel (const QString & service, const QString & appname = QString::null, bool automatics = TRUE) [static]	返回给定 service 的信道，如果没有这个 service，返回 null 字符串
:channels (const QString & service, bool automatics = TRUE) [static]	返回提供 service 的所有应用程序的信道
config (const QString & service) [static]	返回提供 service 的所有应用程序的配置文件名，提供 service 的应用程序可能有定义在 appConfig()文件中的附加的配置

（2）ServiceRequest 类

ServiceRequest 类允许应用程序从其他应用程序请求服务，一个 ServiceRequest类封装了一个服务和发送到服务的消息。它类似于 QCopEnvelope 类，但是使用 service 名，而不是应用程序名。ServiceRequest 类从 QDataStream 类继承，在使用 send()函数发送请求之前应写数据到请求中。

5.8.13 MimeType 类

MimeType 类提供了 MIME 类型信息，一个 MimeType 对象是一个提供 MIME 类型信息的轻量型数据库。它将 mime.types 文件中的 MIME 类型与文件后缀的对应关系读出到链表中，程序从这个链表中可以找到文档文件名后缀对应的 MIME 类型，然后从 AppLnk 对象的链表中查找到 MIME 类型对应

的应用程序。这样，就完成了从文档类型查找打开文档的应用程序的过程。 MimeType 类的成员函数及其功能说明如表 5-30所示。

表5-30 MimeType 类的成员函数及其功能说明

成员函数名	功能说明
MimeType (const QString & ext_or_id)	构建一个 MimeType 对象，通常 ext_or_id 是个 MIME 类型，如果 ext_or_id 以 / 开始或不含有 /，则解释为文件名，且扩展名（如.txt）被用于 MIME 类型
MimeType (const DocLnk & lnk)	构建一个 MimeType 对象
application ()	返回定义了与这个 MIME 类型相关的应用程序的 AppLnk 对象
appsFolderName () [static]	返回装有应用程序定义的 Qtopia 文件夹
bigPixmap ()	返回 MIME 类型的大图标的 QPixmap 对象
description ()	返回 MIME 类型的描述，通常基于与类型相关的 application ()
id ()	返回 MIME 的 ID
pixmap ()	返回 MIME 类型的小图标的 QPixmap 对象
updateApplications () [static]	重装载应用程序定义

每个应用程序启动后会创建 MimeType对象，MimeType类使用 MimeTypeData 类对象保存 MIME 类型数据，使用全局变量 d 来保存。它是 Private 类型的指针，通过 data()函数可访问这个变量。

Private类从 QDict 继承，它将 MimeTypeData 对象组织成字典链表，以 MIME 类型字符串作为键值，通过键值可找到对应的 MimeTypeData对象。

Private类及函数 data()列出如下：

```
MimeType::Private* MimeType::d=0;  //静态变量的初始化
class MimeType::Private : public QDict<MimeTypeData> {
public:
    Private() {}
    ~Private() {}
};

MimeType::Private& MimeType::data()  //返回*d
{
    if ( !d ) {
            d = new Private;
            d->setAutoDelete(TRUE);
            static bool setCleanup = FALSE;  //保证只清除 1 次，清除多次会导致应用程序崩溃
            if ( !setCleanup ) {
            /*在 QApplication 析构中会调用 qAddPostRoutine 例程，通常用来清除应用程序的全局变量空间*/
                qAddPostRoutine( cleanupMime );  //    用来清除 MimeType 类的全局变量空间
                setCleanup = TRUE;
            }
    }
    return *d;
}

static void cleanupMime()
{
    MimeType::clear();
}

void MimeType::clear()
{
   delete d;
   d = 0;
   delete extFor;
   extFor = 0;
```

```
    delete typeFor;
    typeFor = 0;
}
```

MIME 类型数据用 MimeTypeData 类描述，它包括 MIME 类型名字符串、扩展名字符串，以及 MIME 类型对应的应用程序的 AppLnk 对象链表，这样通过 MIME 类型名字符串可查找对应的扩展名字符串及 AppLnk 对象链表。MimeTypeData 类的定义分析如下：

```
class MimeTypeData {
public:
    MimeTypeData(const QString& i) :
        id(i)
    {
            apps.setAutoDelete(TRUE);
    }
    QString id; //MIME 类型名字符
    QString extension; //扩展名字符串
    QList<AppLnk> apps; //MIME 类型对应的应用程序的 AppLnk 对象链表
    //得到 apps 中第 1 个 AppLnk 对象的 name 值，即应用程序的本地名
    QString description()   {..... }
    //得到与应用程序相关的小图标位图
    QPixmap regIcon()   {..... }
    //得到与应用程序相关的大图标位图
    QPixmap bigIcon()   {.....  }

private:
     void loadPixmaps()   {  .....  }  //装载应用程序的图标位图
     .....
};
```

为了查找 MIME 类型及扩展名，MimeType 类还使用了两个全局静态变量*typeFor 和*extFor。其中变量*typeFor 以扩展名字符串作为 Hash 值保存扩展名对应的应用程序类型，变量*extFor 以应用程序类型为 Hash 值保存扩展名字符串链表。

例如：(*typefor)["video/quicktime"]值为“mov qt”，(*typeFor)[mov]=video/quicktime, (*typeFor)[qt] = video/quicktime。

全局静态变量*typeFor 和 *extFor 的声明列出如下：

```
static QMap<QString,QString> *typeFor = 0;
static QMap<QString,QStringList> *extFor = 0;
```

/etc/mime.types 文件和$QTOPIA_PATH/etc/mime.types 文件保存 MIME 类型及支持的扩展名，MimeTypeData 类是对这两个文件内容的描述，或者说是这两个文件的代表。 mime.types 文件部分列出如下：

```
#MIME 类型                  扩展名
text/html               html htm
text/plain               txt asc c cc h hh cpp hpp
text/richtext            rtx
```

在可打开文档的应用程序对应的.desktop 文件中有 MimeType项，用来说明这个应用程序可打开哪些 MIME 类型的文档。这样，从文档扩展名查询 mime.type 文件可知 MIME 类型，通过 MIME 类型可从.desktop 文件中找到对应的应用程序。程序代码通过 MimeType 类、AppLnkSet 类和 AppLnk 来完成这个查找过程。

MimeType 类的成员函数中许多是静态的，这个类的构造函数从 mime.types 文件中读出配置放入全局静态变量*typeFor、*extFor 和 d 中，MimeType 类构造函数的调用层次如图 5-7 所示。

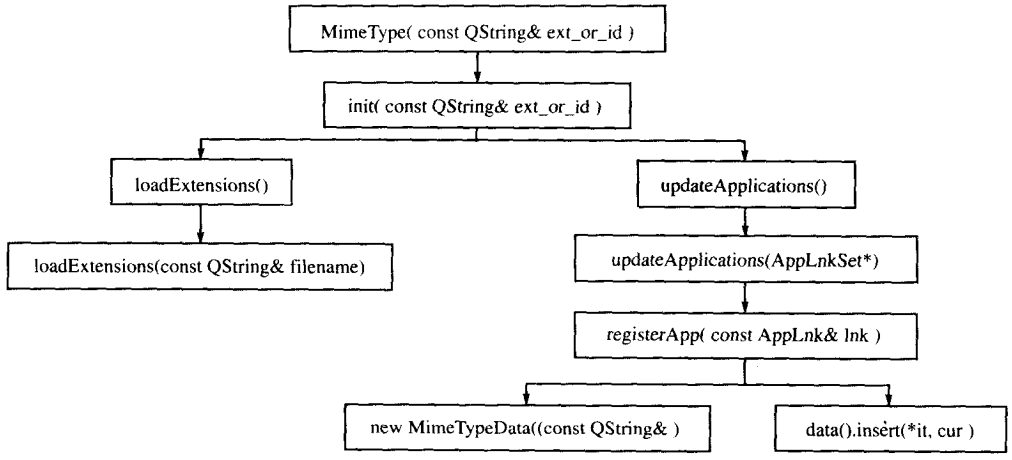


图 5-7　MimeType 类构造函数的调用层次

MimeType 类构造函数分析如下：

```
MimeType::MimeType( const DocLnk& lnk )
{
    init(lnk.type());
}

void MimeType::init( const QString& ext_or_id )
{
    if (ext_or_id[0] != '/' && ext_or_id.contains('/') ) {
          i = ext_or_id.lower();  //转换成小写
    } else {
          loadExtensions();  //装载 mime.types 文件
          int dot = ext_or_id.findRev('.');
          QString ext = dot >= 0 ? ext_or_id.mid(dot+1) : ext_or_id;
          i = (*typeFor)[ext.lower()];
          if ( i.isNull() )  i = "application/octet-stream";
    }

    static bool appsUpdated = FALSE;  //被用来保证只调用一次 updateApplications()
    if ( !appsUpdated ) {
          appsUpdated = TRUE;
          updateApplications();//保存 MimeTypeData 到对象链表 d 中
    }
}
```

函数 loadExtensions()装载 mime.types 文件的信息到链表 extFor 和 typeFor，列出如下：

```
void MimeType::loadExtensions()
{
    if ( !typeFor ) {  //如果 typeFor 不存在
       extFor = new QMap<QString,QStringList>;
       typeFor = new QMap<QString,QString>;
       loadExtensions("/etc/mime.types");
       loadExtensions(QPEApplication::qpeDir()+"etc/mime.types");
    }
}

void MimeType::loadExtensions(const QString& filename)
{
    QFile file(filename);
    if ( file.open(IO_ReadOnly) ) {
       QTextStream in(&file);
```

```
        QRegExp space("[ \t]+");
        while (!in.atEnd()) {
            //读取一行，如字符串为 video/quicktime mov qt
            QString line = in.readLine();
            if (line.isEmpty() || line[0] == '#')
                continue;
            //每行字符串以空格分开，将各个字符串存入字符串链表 tokens 中
            QStringList tokens = QStringList::split(space, line);
            //得到第一个字符串，如 video/quicktime
            QStringList::ConstIterator it = tokens.begin();
            if ( it != tokens.end() ) {
            //id 值如：video/quicktime，++it 后，*it 为 mov
                QString id = *it; ++it;
                // new override old (though left overrides right)
                //从全局变量(*extFor)[]中找到 id 对应的扩展名字符串，如 qt
                QStringList exts = (*extFor)[id];
                QStringList newexts;
                if ( it != tokens.end() ) {
                    exts.remove(*it); //从 exts 中删除*it，如*it 为 mov
                    if ( !newexts.contains(*it) )
                        newexts.append(*it); //newexts 中值为 mov
          //循环后，(*typeFor)[mov]=video/quicktime, (*typeFor)[qt]= video/quicktime
                    while (it != tokens.end()) {
                        (*typeFor)[*it] = id;
                        ++it;
                    }
                }
                (*extFor)[id] = newexts + exts; //字符串链表相加，即其中值为 mov 和 qt
            }
        }
    }
}
```

函数 registerApp 创建并保存 MimeTypeData 到对象链表 d 中，分析如下：

```
void MimeType::registerApp( const AppLnk& lnk )
{
    //从.desktop 对应的 AppLnk 对象中读出 MimeType 项，如 MimeType=text/*
    QStringList list = lnk.mimeTypes();
    for (QStringList::ConstIterator it = list.begin(); it != list.end(); ++it) {
        /* data()[*it]是以 MIME 类型为键值的 MimeTypeData 对象链表，*it 是装有诸如 text/x-vcard 的 Qstring 这样的对象
*/
        MimeTypeData* cur = data()[*it];
        AppLnk* l = new AppLnk(lnk);
        if ( !cur ) {//如果 cur 不存在
            cur = new MimeTypeData( *it );
            //插入键值为*it，值为 cur 的值
            data().insert( *it, cur );
            cur->apps.append(l);
        } else if ( cur->apps.count() ) {
            QString type=*it;
            if ( type.right(2)=="/*")     //如：MimeType=text/*
                type.truncate(type.length()-2);  //去掉字符串末尾的"/*"
            Config binding(serviceBinding("Open/"+type));  //如：Service-Open-text
            QString def;
            if ( binding.isValid() ) {
                binding.setGroup("Service");
                def = binding.readEntry("default");
            }

            if ( l->exec() == def )
                cur->apps.prepend(l);
            else
                cur->apps.append(l);
```

```
        } else {
          cur->apps.append(l);
        }
    } //endfor
}
```

5.8.14 其他类说明

其他类说明如下。

（1）AlarmServer 类

AlarmServer 类允许增加或取消闹钟，应用程序使用 addAlarm()增加闹钟，使用 deleteAlarm()删除闹钟。当闹钟的定时时间到时，特定的 QCop 消息被发送到指定的信道。应该使用这个类，而不是 QTimer 来调度闹钟。因为机器可能是睡眠的，并且需要得到唤醒来使用 Linux 内核。在睡眠时，机器在内核级上尽量减小电池的使用。

（2）Accessory 类

Accessory 类允许 applet插件添加 Accessory 对象到 Qmenu 上，它可以添加单个菜单条目或一个子菜单。当在任务栏上使用真实的状态是浪费的时候， Accessory 类将是有用的。 Accessory 类构造函数创建菜单，并根据 Taskbar 配置文件的 Menu 组中的条目来设置菜单条目，该类的成员函数及其功能说明如表 5-31 所示。

表 5-31 Accessory类的成员函数及其功能说明

成员函数名	功能说明
activated () [signal]	当菜单项被激活时被发送
activated (int) [signal]	当子菜单上的一个菜单项被激活时被发送
iconSet ()	返回显示在菜单上的 QIconSet 对象（代表一组图标）
popup ()	返回子菜单的 QPopupMenu 对象指针，如果子菜单不存在，返回 0
setIconSet (const QIconSet & icon)	设置显示在菜单上的图标
setText (const QString & text)	设置显示在菜单上的文本
text ()	返回显示在菜单上的文本

（3）Global 类

Global 类提供了应用程序范围的全局函数，包括用户界面、文档、文件系统、字链表、限引用字符及硬件相关方面的全局函数。 Global 类的成员函数及其功能说明如表 5-32 所示。

表 5-32 Global 类的成员函数及其功能说明

成员函数名	功能说明
ddWords (const QStringList & wordlist) [static]	加字链表 wordlist 到 QDawg 对象，这个操作较慢，尽量对一个大链表操作一次
addWords (const QString & dictname, const QStringList & wordlist) [static]	添加字链表 wordlist 到 QDawg 对象 dictname
addedDawg () [static]	返回变化的 QDawg 对象，它装有本地化通用字符
applicationFileName (const QString & appname, const QString & filename) [static]	返回给定文件名 filename，应用程序名为 appname 的应用程序的全路径。如果出错，返回 NULL。如果 filename 中含有 /，调用者必须确保这些目录的存在
architecture () [static]	返回设备结构描述的字符串，字符由 / 分隔，从最通用的到最特殊的（如“IBM/PC”）
dawg (const QString & name) [static]	返回给定的 name 的 QDawg 对象，它是一个应用程序特定的字链表，name 不应有 /，如果 name 以 _ 开始，它将是一个仅读的系统字链表

续表

成员函数名	功能说明
decodeBase64 (const QByteArray & encoded) [static]	解码基于 64bit 的数据流 encoded，返回解码的数据流
defaultButtonsFile () [static]	返回 defaultbuttons.conf 文件的名字，这个函数允许 Qtopia 使用 qvfb 提供了 defaultbuttons.conf 文件
deviceId () [static]	返回设备惟一的 ID 号，这个值可以变化，例如在设备被重设置时
encodeBase64 (const QByteArray & origData) [static]	用 base64 映射来编码数据流 origData，并返回编码的数据流
execute (const QString & c, const QString & document = QString::null) [static]	运行应用程序 c，并传递文档 document。较好的访问方法是发送一个 QCop 消息到应用程序的 QPE/Application/appname 信道上
findDocuments (DocLnkSet * folder, const QString & mimefilter = QString::null) [static]	在系统的文档目录中查找所有匹配 mimefilter 的文档，并追加 DocLnk 对象到 folder
fixedDawg () [static]	为当前本地化返回装有通用字的不变的 QDawg 对象
generateUuid () [static]	产生系统范围内惟一的 Uuid
hasKey (int key) [static]	如果 key 在设备上可用的，返回 TRUE。按钮可能是固定功能的（如 Key_Menu）或用户映射的按键，如 Key_F1
hideInputMethod () [static]	隐藏当前输入法，当前输入法还指示在任务栏上，但不再占用屏幕空间并不能与人交互
homeDirPath () [static]	返回被用做当前用户的 home 目录的名字
isAppLnkFileName (const QString & file) [static]	如果 file 是应用程序的名字，返回 TRUE
isDocumentFileName (const QString & file) [static]	如果 file 是一个文档的文件名，返回 TRUE
msleep (unsigned long msecs) [static]	睡眠 msecs 毫秒，它是独立于系统的睡眠
ownerName () [static]	返回设备的拥有者的名字
qtopiaPaths () [static]	返回装有 Qtopia 文件的基目录的链表，这个值是通过环境变量 QPEDIRS 设置的，默认值仅是$QPEDIR
removeWords (const QStringList & wordlist) [static]	从所加的 QDawg 对象中删除 wordlist。如果字是在固定字典中，这些字将被添加到 deleted 字典中
removeWords (const QString & dictname, const QStringList & wordlist) [static]	从命名为 dictname 的 dawg()中删除 wordlist
renameFile (QString from, QString to) [static]	从 form 到 to 重命名文件，操作成功，返回 TRUE
setWeekStartsMonday (bool v) [static]	设置星期开始的日期，如果 v 是 TRUE，那么周将从星期一开始；否则将从星期天开始
void Global::showInputMethod () [static]	显示当前的输入方法
sleep (unsigned long secs) [static]	睡眠 secs 秒，它独立于系统
statusMessage (const QString & message) [static]	显示使用者的状态消息，它经常出现在任务栏一段时间，接着消失
tempName (const QString & fname) [static]	返回适合被用作临时文件的文件名，使用 fname 作为基础
usleep (unsigned long usecs) [static]	睡眠 usecs 微秒，它独立于系统
version () [static]	返回 Qtopia 的版本字符串，形式为：major.minor.patchlevel（例如：“1.2.3”）
weekStartsOnMonday () [static]	如果星期从星期一开始，则返回 TRUE；如果从星期日开始，返回 FALSE
writeHWClock () [static]	写系统时钟到硬件钟中

第 6 章　Qtopia 服务器

Qtopia 服务器是应用程序的管理器，作为一个设置屏幕上的主窗口，它提供服务器窗口，负责启动应用程序，协调管理各个应用程序的运行，打开文档，并显示应用程序或硬件状态等。服务器启动后，PDA 设备屏幕的显示如图 6-1 所示。

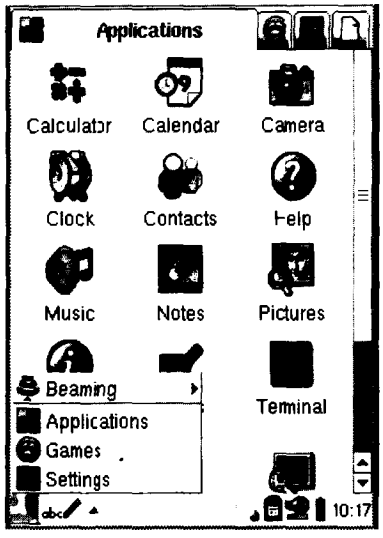


图 6-1　Qtopia 服务器窗口

本章主要说明组成 Qtopia 服务器各个类的功能，分析 Qtopia 服务器的启动过程、如何启动应用程序及插件，以及电源监控、外观设置、应用程序安装与卸载、 AppServices 等与系统管理相关的小应用程序。

6.1　Qtopia 服务器特定的类

Qtopia 服务器窗口由 Lancher 类对象组成，Lancher 窗口部件由 LancherTabWidget 类和 TaskBar 类窗口部件组成。LancherTabWidget 窗口分为 TabBar 和应用程序图标视图，TabBar 切换 4 个不同的视图，即应用程序视图、游戏视图、设置视图和文档视图。视图中放置图标，每个图标代表一个 LauncherItem 类对象。LauncherItem 类包含 AppLnk 对象指针，AppLnk 与一个具体的应用程序或打开文档的应用程序连接起来。通过用鼠标点击图标，即可打开应用程序或文档。 Qtopia 服务器窗口的 4 个视图如图 6-2 所示。

组成 Qtopia 服务器的特定的类都是从 Qt/Embedded 或 Qtopia 的通用类继承而来，它们只在 Qtopia 服务器中使用，本节对这些类的功能分别进行说明。

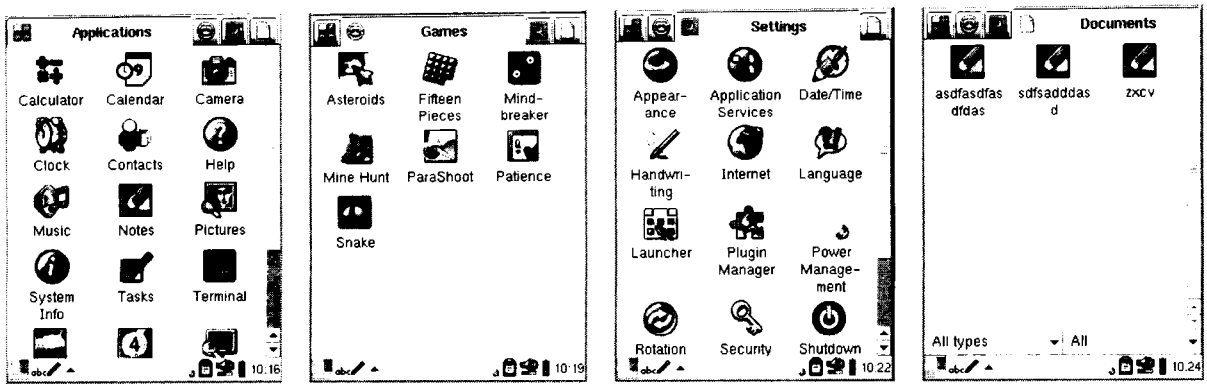


图 6-2 Qtopia 服务器窗口的 4 个视图

6.1.1 ServerApplication 类

ServerApplication 类是 Qtopia 服务器专用的，它继承于 QPEApplication 类。该类的构造函数设置位图缓冲限制，定时检查电池状态、内存及存储空间，并设置键盘过滤等。通过 QPEApplication 父类对应用程序的事件、运行及信道进行全局性管理。 QPEApplication 类在第 4 章中已分析。ServerApplication 类的成员函数及其功能说明如表 6-1 所示。

表 6-1 ServerApplication 类的成员函数及其功能说明

成员函数名	功能说明
screenLocked()	如果屏幕锁住，返回 TRUE；否则返回 FALSE
lockScreen(bool b)	设置是否锁住屏幕
login(bool at_poweron)	如果还没有登录注册，则需要登录注册。它弹出一个全屏的注册对话框，直到注册密码正确后设备才可用，这个对话框也才消失
haveFeature(const QString &f)	判断服务器是否具有 f 字符串描述的功能特征
togglePower()	运行 shell 命令 apm –suspend 挂起电源，如果从屏保中唤醒状态，发消息点亮背景灯
toggleLight()	发消息触发背景灯状态从亮到暗或从暗到亮改变
showSafeMode()	显示安全模式对话框，并发消息运行 pluginmanager 应用程序
clearSafeMode()	清除安全模式设置写配置文件
shutdown()	关闭服务器应用程序
restart()	重新启动服务器应用程序
checkMemory()	检查内存，发消息恢复内存
memoryMonitor()	默认的内存监视器，检查出现的页数及空闲内存是否达到警戒线，发消息恢复内存
evalMemory()	设置内存警戒线
readMemInfo(int *freeMem)	从/proc/meminfo 文件中读取内存信息

6.1.2 Server 类

Server 类从 QWidget 类继承，它代表服务器的窗口系统，并且通过 Launcher 类来管理服务器 UI 界面。StorageInfo 类得到保存空间信息，TempScreenSaverMonitor 类管理屏幕保护状态，DocumentList 类管理文档链表， AppLauncher 类管理应用程序的启动运行， IrServer 类管理红外传输服务，PackageHandler 类进行安装包处理。该类还处理与 Qtopia Desktop 同步过程中的消息，Sever 类的成员函数及其功能说明如表 6-2 所示。

表 6-2 Server 类的成员函数及其功能说明

成员函数名	功能说明
mkdir(const QString &path)[static]	若 path 对应有目录存在，返回 TRUE；否则创建目录。创建成功，返回 TRUE；失败返回 FALSE
show()	显示服务器窗口
soundAlarm()[static]	打开蜂鸣器设备，设置声音
setKeyboardLayout(const QString &kb) [static]	设置按键映射，重载 QWSServer 服务器中的按键映射
systemMsg(const QCString &, const QByteArray &)[slot]	处理与桌面 Qtopia 的数据同步消息，如得到版本号、握手信息、设置鼠标及同步操作的启/停等
receiveTaskBar(const QCString &msg, const QByteArray &data) [slot]	处理任务栏的消息，如报警、设置 LED 并重装应用程序。
terminateServers()[slot]	关闭使用 socket 的传输服务器
pokeTimeMonitors()[slot]	通知所有的时间监视器时间发生变化
recoverMemory(ServerApplication::MemState) [slot]	根据内存状态，关闭打开文档的应用程序及最近打开的应用程序，从而释放内存。预装的应用程序不能关闭
activate(const DeviceButton* button, bool held) [slot]	发出按键按下或按住消息
syncConnectionClosed(const QHostAddress &) [slot]	同步连接关闭，关闭同步对话框
applicationRaised(const QString &app) [slot]	升起 app 应用程序的窗口
applicationLaunched(int pid, const QString &app) [slot]	设置应用程序 app 的状态为 Launching，如果运行的应用程序超过 6 个，则关闭应用程序。预装的应用程序不能关闭
applicationTerminated(int pid, const QString &app) [slot]	设置应用程序 app 的状态为 Terminated，从全局状态链表中删除 pid 对应的项
applicationConnected(const QString &app) [slot]	设置应用程序 app 的状态为 Running
storageChanged()[slot]	存储设备发生变化，重设置对存储设备的状态显示 UI，重新扫描文档得到文档链表
cancelSync()[slot]	发出消息取消同步操作，删除同步操作对话框
finishedQueuedRequests() [slot]	完成排队的请求，进入屏保

6.1.3 Launcher 类

Launcher 类从 QMainWindow 类继承，它是 PDA 用户界面接口。其成员函数 createGUI 创建任务栏 TaskBar 对象放在屏幕的下边，创建 LauncherTabWidget 对象来分类显示应用程序及文档图标。服务器在调用 Launcher 类的成员函数 createGUI()后，生成的 PDA 用户界面窗口。Lancher 类的各个窗口部件类分布如图 6-3 所示。

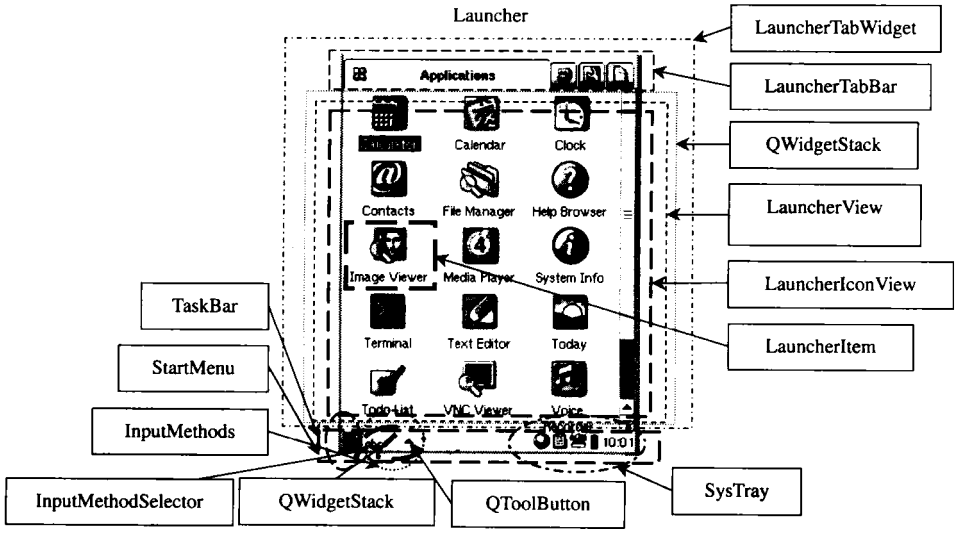


图 6-3 Launcher 类的各个窗口部件类分布

Launcher 类的成员函数及其功能说明如表 6-3 所示。

表 6-3 Launcher 类的成员函数及其功能说明

成员函数名称	功能说明
createGUI()	创建任务栏 TaskBar 对象放在屏幕的下边，创建 LauncherTabWidget 对象来分类显示应用程序及文档图标，安装事件过滤器，创建管理各个文档图标的视图
showGUI()	延迟显示用户界面
showTab(const QString& id)	显示 id 字符串对应的 Tab
select(const AppLnk *appLnk)	运行鼠标点击选中图标的应用程序
properties(const AppLnk *appLnk)	显示选中图标的应用程序的属性对话框
typeAdded(...)	增加 Tab 及对应的应用程序图标视图，把新添加的 Tab 放在第 1 序号，并显示对应的 Tab 及视图
typeRemoved(const QString& type)	移去一个 Tab 及其对应的应用程序图标视图，并清除视图的所有的应用程序图标
applicationAdded(...)	在指定类型的视图中增加一个应用程序图标，并注册 MimeType
applicationRemoved(...)	在指定类型的视图中删除一个应用程序图标
allApplicationsRemoved()	删除所有视图中的所有应用程序图标
documentAdded(const DocLnk& doc)	在文档视图中增加一个文档的图标
allDocumentsRemoved()	在文档视图中增加所有文档的图标

6.1.4 LauncherTabWidget、LauncherTabBar 和 LauncherTab 类

LauncherTabWidget、LauncherTabBar 和 LauncherTab 类说明如下。

（1）LauncherTabWidget 类

LauncherTabWidget 类从 QVBox 类继承，它是包括应用程序视图、文档视图与 TabBar 的窗口部件。其构造函数创建了 LauncherTabBar、QWidgetStack 和 LoadingWidget 对象，其中 QWidgetStack 用于将不同种类的视图（如应用程序、游戏、设置和文档视图）叠放在一起。LauncherTabWidget 类的成员函数及其功能说明如表 6-4 所示。

表 6-4 LauncherTabWidget 类的成员函数及其功能说明

成员函数名	功能说明
updateDocs(AppLnkSet* docFolder)	更新文档视图中的文档连接
setBusy(bool on)	设置当前视图忙状态为 on
currentView(void)	返回当前视图的 LauncherView 对象指针
newView(const QString&, const QPixmap& pm, const QString& label)	创建新的视图 LauncherView 对象和 LauncherTab 对象，并根据配置文件设置窗口部件
deleteView(const QString& id)	删除 id 序号的 Tab 及对应视图
setTabViewAppearance(LauncherView *v, Config &cfg)	读出配置文件 cfg 的相应配置项，设置视图列表或缩略图显示方式，并设置字体及背景颜色等
setTabAppearance(LauncherTab *tab, Config &cfg)	读出配置文件 cfg 的相应配置项，设置 Tab 字体及背景颜色等
view(const QString &id)	返回 id 序号 Tab 对应的视图 LauncherView 对象指针
docView()	返回文档视图的 LauncherView 对象指针
createDocLoadingWidget()	创建 DocLoadingWidget 对象
setLoadingWidgetEnabled(bool v)	如果 v 与 LoadingWidget 的 enabled()状态不一致，则设置状态，并升起当前视图窗口
setLoadingProgress(int percent)	设置 LoadingWidget 窗口部件的进度条计数为 percent
launcherMessage(const QCString &, const QByteArray &)[slot]	处理信道 QPE/Launcher 上的 receive 消息，如设置视图列表或缩略图显示方式、字体及背景颜色等

（2）LauncherTabBar 类

LauncherTabBar 类从 QTabBar 类继承，当 TabBar 被选中时，发出 selected（int）信号，连接的槽函数 layoutTabs()设置并绘制 TabBar 的外观等。它还使用下面的链表保存 LauncherTab 对象：

```
QList<LauncherTab> items;
   QDict<LauncherTab> tabs;
```

LauncherTabBar 类的成员函数及其功能说明如表 6-5 所示。

表 6-5 LauncherTabBar 类的成员函数及其功能说明

成员函数名	功能说明
insertTab(LauncherTab *t, int index)	将 t 插入到 QTabBar 序号为 index 的地方，并插入到链表中
removeTab(QTab *tab)	从 QTabBar 对象中删除 tab，并在链表中删除 tab
*currentView()	得到当前的 TabBar 对应的 Tab 的 LauncherView 对象
*currentLauncherTab()	得到当前的 TabBar 对应的 Tab 的 LauncherTab 对象
*launcherTab(const QString &id)	得到序号为 id 的 TabBar 对应的 Tab 的 LauncherTab 对象
nextTab()	设置当前的 Tab 为下一个 Tab
prevTab()	设置当前的 Tab 为前一个 Tab
showTab(const QString& id) [slot]	设置当前的 TabBar 为序号 id 的 Tab
layoutTabs()[slot]	设置并绘制 TabBar 的外观等
paint(QPainter *p, QTab *t, bool f)	绘制 TabBar 的图标及颜色等
paintLabel(QPainter p, const QRect& br, QTab* t, bool has_focus)*	绘制标签的图标及字体

（3）LauncherTab 类

LauncherTab 类从 QTab 类继承，它提供了 LauncherTabBar 类中用到的结构，其中包含 LauncherView 对象指针。

（4）LauncherIconView 类

LauncherIconView 类从 QIconView 类继承，根据屏幕大小计算并设置网格，在图标视图中显示应用程序或文档的图标。LauncherIconView 类的成员函数及其功能说明如表 6-6 所示。

表 6-6 LauncherIconView 类的成员函数及其功能说明

成员函数名	功能说明
busyItem()	返回正忙的 QIconViewItem 对象
busyPixmap()	返回正忙的 QPixmap 对象
setBigIcons(bool bi)	根据参数 bi 决定是否设置为大图标
updateCategoriesAndMimeTypes()	更新视图中每个条目的 MIME 类型
setBusy(bool on)	如果参数 on 为 TRUE，设置视图中当前图标条目为忙，并插放动画
inKeyEvent()	如果有按键输入，返回 TRUE；否则返回 FALSE
keyPressEvent(QKeyEvent* e)	按键事件处理
addItem(AppLnk* app, bool resort=TRUE)	在视图中加入图标条目 LauncherItem 对象，根据参数 TRUE 来决定是否重排序
removeLink(const QString& linkfile)	删除 linkfile 对应的 QIconViewItem 对象
mimeTypes()	返回视图中所有 LauncherItem 对象对应的 MIME 类型链表
clear()	清除视图及 MIME 类型链表
addCatsAndMimes(AppLnk* app)	增加 app 对应的 MIME 类型到 MIME 类型链表
setBackgroundPixmap(const QPixmap &pm)	设置视图背景画为 pm

续表

成员函数名	功能说明
setBackgroundColor	设置视图背景颜色
drawBackground(QPainter *p, const QRect &r)	绘制视图背景
setItemTextPos(ItemTextPos pos)	设置图标条目的文本位置
hideOrShowItems(bool resort)	更新视图中图标的显示，并根据 resort 来决定是否重排序
setTypeFilter(const QString& typefilter, bool resort)	设置类型过滤器
setCategoryFilter(int catfilter, bool resort)	设置 Category(服务器窗口视图的类型)过滤器
setSortMethod(SortMethod m)	按名字、时间或类型设置排序方法
compare(const AppLnk* a, const AppLnk* b)	返回 a 与 b 的比较结果

6.1.5 LauncherView 和 LauncherItem 类

LauncherView 和 LauncherItem类说明如下。

（1）LauncherView 类

LauncherView 类从 QVBox 类继承，其构造函数创建 LauncherIconView 对象，并连接了 LauncherIconView对象的信号。该类承载 LauncherIconView对象的窗口部件，管理 LauncherIconView 对象。

（2）LauncherItem类

LauncherItem 类从 QIconViewItem 类继承，它代表了视图中每个图标对应条目具有的各种功能的集合，并保存 AppLnk 对象指针。LauncherItem 类的成员函数及其功能说明如表 6-7 所示。

表 6-7 LauncherItem类的成员函数及其功能说明

成员函数名	功能说明
*appLnk()	返回图标条目对应的 AppLnk 对象
*takeAppLnk()	返回图标条目对应的 AppLnk 对象，并将类内部使用的 AppLnk 对象指针赋 0
animateIcon()	若是当前选中的图标条目，进行动画高亮处理
resetIcon()	重设置图标条目对应的图标
compare (QIconViewItem * i)	将当前的图标条目的 AppLnk 与图标条目 i 进行比较，返回比较结构
paintItem(QPainter *p, const QColorGroup &cg)	绘制图标条目

6.1.6 AppLauncher 和 QuickLauncher 类

AppLauncher 类和 QuickLauncher 类说明如下。

（1）AppLauncher 类

AppLauncher 类从类 QObject 继承，用于从 QPE/System 信道和 QPE/Server 信道接收消息，启动应用程序或升起窗口部件，并且运行 quicklauncher 应用程序。AppLauncher 类的成员函数及其功能说明如表 6-8 所示。

表 6-8 AppLauncher 类的成员函数及其功能说明

成员函数名	功能说明
isRunning(const QString &app)	从运行的应用程序链表中查找 app 的名字，如果找到且 pid 在链表中，表示应用程序在运行，返回 TRUE；否则返回 FALSE
running()	返回运行的应用程序链表
delayMessages(bool b)[static]	延时调用 receive()处理排队的消息

续表

成员函数名	功能说明
pidForName(const QString &appName)	从运行的应用程序链表中查找 appName 应用程序对应的 pid，返回 pid
criticalKill(const QString &app)[slot]	杀死应用程序 app，并发出 terminated(pid, app)信号
received(const QCString& msg, const QByteArray& data) [slot]	接收并处理 execute 及 raise 消息，调用函数 execute 运行应用程序，并将应用程序加入到运行的应用程序链表中
createQuickLauncher()[slot]	运行 bin/quicklauncher 应用程序
processExited()[slot]	发送消息的应用程序退出，并从运行的应用程序链表中清除这个应用程序

（2）QuickLauncher 类

QuickLauncher 类从 QObject 继承，其构造函数 QuickLauncher()创建 QPE/QuickLauncher 信道。QuickLauncher 类连接处理这个信道上的消息，由这个信道上的消息来决定运行启动哪个应用程序。该类还提供静态函数 exec (int argc, char **argv)用来显示已存在的应用程序的主窗口。

6.1.7 LoadingWidget 和 DocumentList 类

LoadingWidget 和 DocumentList 类说明如下。

（1）LoadingWidget 类

LoadingWidget 类从 LauncherView 类继承，用来装载文档视图，其构造函数创建进度条及显示等待图标的标签等。在打开文档视图时，会显示装载进度条，装载后显示文档视图。

（2）DocumentList 类

DocumentList 类创建 AppLnkSet 对象保存所有在应用程序目录下的应用程序连接信息（图标位图、应用程序名和 MimeType 等）、创建 DocumentListPrivate 对象存储文档路径、保存空间信息，并使用 DocLnkSet 对象保存文档连接信息。DocumentList 类的成员函数及其功能说明如表 6-9 所示。

表 6-9 DocumentList类的成员函数及其功能说明

成员函数名称	功能说明
reloadDocLnks()	重新扫描文档路径下的文档文件，并加入到文档图标视图中
linkChanged(QString arg)	如果是应用程序连接发生变化，重装所有应用程序连接。如果是一个已存在文档连接，则用新连接替换旧连接；如果是一个新的文档连接，加上新的文档连接
restoreDone()	重新装载所用的应用程序和文档连接到图标视图中
sendAllDocLinks()	在 QPE/Desktop 信道上发送文档需要的所有图标消息，然后发送所有的文档连接消息
sendDocLnk(DocLnk *doc)	在 QPE/Desktop 信道上发送文档连接消息
sendIcon(...)	在 QPE/Desktop 信道上发送文档需要的图标消息
reloadAppLnks()	重新装载所用的应用程序连接到图标视图中

6.1.8 TaskBar 类

TaskBar 类从 QHBox 类继承，其构造函数创建 StartMenu 显示“开始”菜单；创建 InputMethods 对象来显示各种输入法；创建 RunningAppBar 对象用来显示应用程序状态；创建 Wait 对象显示应用程序正在运行的图标，并阻塞对这个应用程序的鼠标事件等，以防止用户操作打开应用程序；还创建 SysTray 对象显示 applet 小应用程序。

TaskBar 类中存有 Wait、InputMethods、SysTray、RunningAppBar、QWidgetStack、LockKeyState 和 StartMenu 对象指针。

TaskBar类的成员函数及其功能说明如表 6-10 所示。

表 6-10 TaskBar 类的成员函数及其功能说明

成员函数名	功能说明
launchStartMenu()	弹出“开始”菜单
refreshStartMenu()	删除并重新装载“开始”菜单
startWait()[slot]	显示等待图标，开始等待应用程序的运行完成
stopWait()[slot]	停止等待，关闭等待图标
clearStatusBar()[slot]	清除状态消息标签，显示应用程序状态标签和系统托盘
toggleNumLockState()[slot]	切换键盘 NumLock 键打开/关闭状态
toggleCapsLockState()[slot]	切换键盘 CapsLock 键打开/关闭状态
toggleSymbolInput()[slot]	切换 Unicode 符号输入法打开/关闭状态
calcMaxWindowRect()[slot]	计算并设置显示最大窗口
setStatusMessage(const QString &text)	在任务栏上设置并显示状态消息
receive(const QCString &msg, const QByteArray &data) [slot]	根据消息在任务栏上显示状态消息、装载/卸载输入法并重装载 applet 小应用程序

6.1.9 StartMenu 和 StartPopupMenu 类

StartMenu 和 StartPopupMenu 类说明如下。

（1）StartMenu 类

StartMenu 类从 QLabel 类继承，它创建并管理主菜单，其构造函数设置菜单开始项图标并更新菜单项。StartMenu 类的成员函数及其功能说明如表 6-11 所示。

表 6-11 StartMenu 类的成员函数及其功能说明

成员函数名	功能说明
refreshMenu()	更新菜单，从 Taskbar 配置文件的 Menu 组中读出配置，并删除所有存在的菜单项重新装载。对于不存在的菜单项，根据配置项进行创建
launch()[slot]	菜单上鼠标按下事件的处理函数，如果菜单已可见，则隐藏菜单；如果菜单不可见，则弹出菜单
createMenu()[slot]	创建并装载菜单
itemSelected(int id) [slot]	菜单信号 activated(int)的槽函数，发出 tabSelected(tabs[id])信号，触发打开 id 序号的 Tab，或者运行对应的应用程序

（2）StartPopupMenu 类

StartPopupMenu 类从 QPopupMenu 类继承，其重载函数 keyPressEvent(QKeyEvent *e) 对鼠标事件进行了过滤。

6.1.10 InputMethodSelector 和 InputMethods 类

InputMethods 类是输入法的窗口部件，其中包含 InputMethodSelector 类对象的指针，用于加载输入法插件，一般有手写、键盘输入和选择板几种输入法插件。InputMethodSelector 和 InputMethods 类窗口部件的位置如图 6-4 所示。

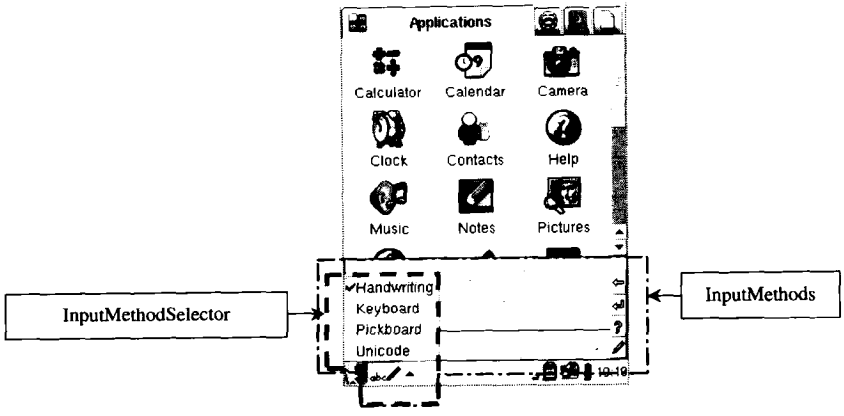


图 6-4 InputMethodSelector和 InputMethods类窗口部件的位置

（1）InputMethods类

InputMethods类从 QWidget类继承，它提供了输入法插件的装/卸载，以及输入法的显示与隐藏，该类的成员函数及其功能说明如表 6-12 所示。

表 6-12 InputMethods类的成员函数及其功能说明

成员函数名	功能说明
InputMethods(QWidget *, IMType=Any)	构造函数，创建 InputMethodSelector 对象，装载输入法，创建 QPE/IME 信道
inputRect()	返回当前输入法的矩形区域 QRect 对象
shown()	如果当前输入法的窗口部件显示，返回 TRUE；否则返回 FALSE
currentShown()	返回当前显示的输入法的名字
unloadInputMethods()	卸载所有的输入法插件
loadInputMethods()	装载各种输入法
showInputMethod(const QString& name)[slot]	显示并激活名为 name 的输入法
showInputMethod()[slot]	激活当前输入法
hideInputMethod()[slot]	隐藏当前输入法

（2）InputMethodSelector类

InputMethodSelector类从 QHBox类继承，它显示输入法工具按键及状态按钮，用来选择输入法，该类的成员函数及其功能说明如表 6-13所示。

表 6-13 InputMethodSelector 类的成员函数及其功能说明

成员函数名	功能说明
InputMethodSelector(QWidget *parent)	创建 QWidgetStack 对象用于在任务栏上放置各种输入法的图标，创建 QToolButton 对象用于在任务栏上放置一个带“^”图标的工具按钮，用来弹出菜单
add(const InputMethod &im)	将输入法 im 加入到输入法链表中，将输入法的状态位图加入到 InputMethod 对象和 QWidgetStack 对象中
clear()	清除所有的输入法，清除输入法链表及对应窗口部件
current()	返回当前输入法的 InputMethod 对象指针
showInputMethod(bool)	显示输入法
setInputMethod(const QString &s)	升起名字 s 的输入法的窗口部件
sort()	输入法状态图标在 QWidgetStack 对象中按在输入法链表的次序进行排序
count()	返回输入法的总数
filterAgainst(const QString &n)	如果当前输入法名为 n，使输入法失效，并隐藏输入法状态图标
activateCurrent(bool on)[slot]	如果 on 为 TRUE，显示当前输入法；为 FALSE，隐藏当前输入法
showList()[slot]	显示输入法菜单，并激活当前输入法

6.1.11 RunningAppBar 和 SysTray 类

RunningAppBar类是应用程序运行状态图标，SysTray 类列出了系统使用的一些小应用程序的窗口部件，它们在任务栏上的位置如图 6-5 所示。

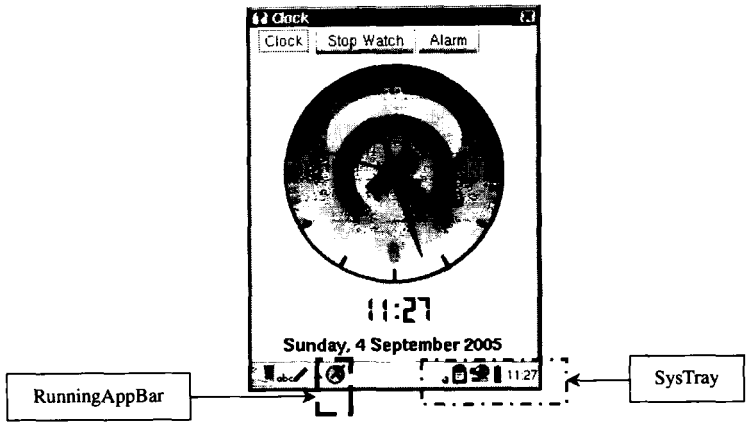


图 6-5 RunningAppBar和 SysTray 类在任务栏上的位置

（1）RunningAppBar 类

RunningAppBar 类从 QFrame 继承，它增加应用程序运行的状态标签到任务栏上，该类的成员函数及其功能说明如表 6-14 所示。

表 6-14 RunningAppBar 类的成员函数及其功能说明

成员函数名	功能说明
RunningAppBar(QWidget* parent)	构造函数，得到 QPE/System 信道，并连接到槽函数 received 在任务栏上显示或隐藏应用程序状态图标
sizeHint()	返回应用程序的图标的大小 QSize 对象
addTask(const AppLnk& appLnk)	加 appLnk 到 appList 链表中，并更新应用程序图标
removeTask(const AppLnk& appLnk)	从 appList 链表中删除 appLnk，并更新应用程序图标

（2）SysTray 类

SysTray 类从 QFrame 类继承，它是系统“托盘”。用于在任务栏右侧上放上小应用程序（applet）的图标并可点击激活小应用程序，这些应用程序多用来显示系统硬件的相关状态或设置硬件，该类的成员函数及其功能说明如表 6-15 所示。

表 6-15 SysTray 类的成员函数及其功能说明

成员函数名	功能说明
SysTray(QWidget *parent)	构造函数，它装载小应用程序，在托盘上显示小应用程序图标
addApplets()	创建 PluginLoader 对象，加载 applets 类型插件到一个链表 appletList 中，并显示 applet
clearApplets()	从链表 appletList 中清除 applet，并删除 PluginLoader 对象

6.1.12 服务器的其他类说明

服务器的其他类说明如下。

（1）Wait 类

Wait 类从 QWidget 类继承，是在应用程序运行时显示的图标，一般为“沙漏”形状图标，它提供了函数 setWaiting（bool w） 来显示（在 w 为 TRUE 时）或隐藏（在 w 为 FALSE 时）图标。

（2）SafeMode 类

SafeMode 类从 QWidget 继承，用于表示机器运行在安全模式。它提供了插件管理器、重启动机器和帮助等简单功能，方便于调试机器。该类的成员函数及其功能说明如表 6-16所示。

表 6-16 SafeMode 类的成员函数及其功能说明

成员函数名	功能说明
mousePressEvent(QMouseEvent *)	创建弹出菜单、插入菜单项和插件管理器、重启动、帮助并连接槽函数 action
action(int i)[slot]	根据选项 i 打开插件管理器、重启动机器或打开帮助文档

（3）TempScreenSaverMonitor 类

TempScreenSaverMonitor 类的成员函数 setTempMode 将每个应用程序 pid 和对应的屏幕保护方式存入数组中，并定时根据当前应用程序的状态发消息设置屏幕保护方式。

（4）SysFileMonitor 类

从 QObject 类继承，其构造函数 SysFileMonitor 创建 QPE/Stabmon 信道，SysFileMonitor 类对象从这个信道上接收并处理 suspendMonitor()消息（停止定时器）和 restartMonitor()消息（启动定时器，间隔为 2 s）。

SysFileMonitor 类重载定时事件 timerEvent()，实现对文件 /var/run/stab、/var/state/pcmcia/stab、/var/lib/pcmcia/stab 和/etc/mtab 的监控，当这几个文件修改时间发生变化时，向信道 QPE/Card 发出 stabChanged() 或 mtabChanged() 消息。stabChanged() 消息表示有卡插入/拔出的信息记录在 stab 文件中，mtabChanged()表示有文件系统挂接/卸载的信息记录在 mtab 文件中。

（5）StorageMonitor 类

StorageMonitor 类从 QObject 类继承，其构造函数创建 StorageInfo 对象。通过 StorageInfo 对象可得到各个文件系统挂接点信息。同时连接信道 QPE/System，处理 outOfDiskSpace（QString）消息（弹出警告对话框并发消息清除空间）和 checkDiskSpace() 消息（检查可用的空间）。

（6）FirstUse 类

FirstUse 类从 QDialog 类继承，在设备第 1 次启动时，它提供对话框供用户设置语言和时区并写入到配置文件中。

（7）IrServer 类

IrServer 类从 QObject 类继承，它只有构造函数，装载了 obex 插件。

（8）PackageHandler 类

PackageHandler 类从 QObject 类继承，用来对应用程序包的安装及卸载进行管理，该类的成员函数及其功能说明如表 6-17 所示。

表 6-17 PackageHandler 类的成员函数及其功能说明

成员函数名	功能说明
PackageHandler(QObject *parent, char* name)	构造函数，创建 QPE/Package 信道，并将接收的消息连接到槽函数 qcopMessage
qcopMessage(const QCString &msg, const QByteArray &data) [slot]	根据接收到的消息安装或卸载软件包、增加或删除软件包
installPackage(const QString &package, const QString &dest)	使用命令 ipkg 安装包到目的地 dest
removePackage(const QString &package)	使用命令 ipkg remove 卸载软件包 package

续表

成员函数名	功能说明
addPackageFiles(const QString &location, const QString &listfile)	拷贝文件 listfile 到 location 位置
addPackages(const QString &location)	将包中文件拷贝到 location 位置
cleanupPackageFiles(const QString &listfile)	删除文件 lisgfile
cleanupPackages(const QString &location)	删除位于 location 处当前软件包的所有文件
prepareInstall(const QString& size, const QString& path)	判断 path 路径所在的硬盘空间是否大于 size，即是否有可安装的空闲空间
sendReply(const QCString& msg, const QString& arg)	在信道上发送 msg 消息，arg 是 msg 消息的参数

6.2 Qtopia 服务器启动过程

6.2.1 Qtopia 服务器的 main 函数

main 函数是 Qtopia 服务器的入口函数，从其开始一步步地建立服务器运行环境及服务器窗口系统，最后进行主事件循环。main 函数的调用层次如图 6-6 所示。

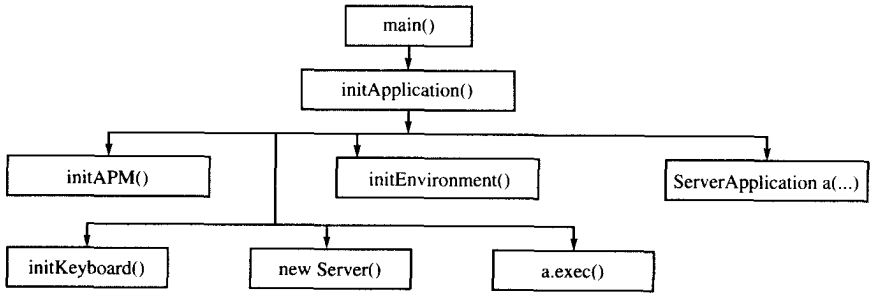


图 6-6 Qtopia 服务器 main 函数的调用层次

从图中可看出，服务器功能主要通过 ServerApplication 和 Server类来实现。Server Application 类对整个服务器应用程序的事件、信道及运行进行全局性管理，Server 类实现了服务器窗口系统的 UI 界面。

Qtopia 服务器的主函数 main()（在 src/server/main.cpp 中）分析如下：

```
int main( int argc, char ** argv )
{
    signal( SIGCHLD, SIG_IGN );//忽略信号
    int retVal = initApplication( argc, argv );//初始化应用程序

    // 如果我们被请求重新启动
    if ( ServerApplication::doRestart ) {
        for ( int fd = 3; fd < 100; fd++ )
            close( fd ); //关闭文件描述符
        execl( (QPEApplication::qpeDir()+"bin/qpe").latin1(), "qpe", 0 );//重运行 qpe
    }
    ...
    return retVal;
}
```

函数 initApplication 读出配置设置环境变量、建立时区、设置开机 LOGO、创建 Sever 实例、设置

背景灯，配置键盘参数等，并进入主事件循环。代码如下：

```
int initApplication( int argc, char ** argv )
{
#ifdef QT_QWS_CASSIOPEIA
    //针对特定设备 Cassiopeia，初始化时间、环境变量、并建立目录挂接特定的文件系统
    initCassiopeia();
#endif

#ifdef QPE_OWNAPM
    initAPM();//取消电源管理
#endif

#ifdef QT_DEMO_SINGLE_FLOPPY
    initFloppy();// 挂接/proc，设置 HOME、QTDIR、QWS_SIZE 环境变量
#endif

    initEnvironment(argc, argv);  // 从配置文件中读出配置，并设置环境变量

    //开机时不会闪烁
#ifdef QWS
    QWSServer::setDesktopBackground( QImage() );  //设置桌面背景画面
#endif
    ServerApplication a( argc, argv, QApplication::GuiServer ); //      构造服务器应用程序
    cleanup();  //在临时目录 /tmp 中创建目录 qcop-msg-* ，并清空目录
    refreshTimeZoneConfig(); // 更新时区配置文件
    initBacklight();  // 在 QPE/System  信道上发送消息 setBacklight(int)
    initKeyboard();  // 从配置文件 qpe 中读取键盘配置，设置键盘参数

#if defined(QTOPIA_PHONE)
    SplashScreen *splash = new SplashScreen;
    splash->splash(Resource::findPixmap("splash"));  //播放动画屏幕
#else
    //在 Windows 下不需要进行第 1 次使用的设置
# if defined(Q_OS_UNIX)
    if ( firstUse() ) {//如果是第 1 次使用，将进行设置时区及语言等操作，然后重新启动
        a.restart();
        return 0;
    }
# endif
#endif

#if defined(QT_QWS_LOGIN)
    for( int i=0; i<a.argc(); i++ )
      if( strcmp( a.argv()[i], "-login" ) == 0 ) { //      不需要使用 tr()翻译字符串
        QDMDialogImpl::login( );
        return 0;
      }
#endif

    Server *s = new Server();
    s->connect(&a, SIGNAL(recoverMemory(ServerApplication::MemState)),
           SLOT(recoverMemory(ServerApplication::MemState)));

    ServerApplication::login(TRUE);// 显示 login 对话框，检查密码是否正确

#ifndef QPE_SYSTEM_SYSFILEMONITOR
    //当个变化发生时，设备应该发送 QCop 消息
    //创建信道 QPE/Stabmon，SysFileMonitor 对象监视有否卡插入，并发出消息
    (void)new SysFileMonitor(s);
#endif

#ifdef QWS
    //创建“ QPE/Network ”和“ QPE/Card ”信道，开始 2 s 的定时器
```

```
    Network::createServer(s);
#endif

//显示s
#if defined(QTOPIA_PHONE)
    splash->setReplacement(s); //用s替代来显示
#else
    s->show();
#endif

    //建立alarm服务器，从文件中读出定时器时间数据，并设置定时器
    AlarmServer::initialize();

    int rv = a.exec();
    qDebug("exiting...");
    delete s;

    return rv;
}
```

6.2.2 ServerApplication 类构造函数

ServerApplication 类的构造函数（在 src/server/serverapp.cpp 中）分析如下：

```
ServerApplication::ServerApplication( int& argc, char **argv, Type t )
    : QPEApplication( argc, argv, t ), userInfoBox( 0 )
{
#ifdef CUSTOM_SOUND_INIT
    CUSTOM_SOUND_INIT;
#endif

#if defined(QPE_MEMALERTER_IMPL)
    initMemalerter();//初始化内存报警器
#endif

    if (!features)
     features = new QStringList;  //用于记录特征的字符串链表
    features->clear();

    // App/DocLnks 可能导致许多缓冲的位图，因此限制位图缓冲
    QPixmapCache::setCacheLimit(512);
#ifdef QTOPIA_PHONE
    PhoneLauncher::initExportedBackgrounds();
    Config btnCfg(Global::defaultButtonsFile(), Config::File);
    btnCfg.setGroup("Device");
    QStringList ilist = btnCfg.readListEntry("Input", ',');
    if (!ilist.isEmpty()) {
        *features += ilist;
    } else {
        if (Global::mousePreferred()) {//是触摸屏还是键盘
# ifdef QPE_NEED_CALIBRATION
            features->append("Calibrate");
# endif
            features->append("Touchscreen");
        } else {
            features->append("Keypad");
        }
    }
#else
# ifdef QPE_NEED_CALIBRATION
    features->append("Calibrate");
# endif
    features->append("Touchscreen");
```

```
#endif

    QTimer *timer = new QTimer( this );//定时检查电池状态
    connect( timer, SIGNAL(timeout()), this, SLOT(psTimeout()) );
    timer->start( 10000 );
    ...
//检查内存
#if !defined(QPE_MEMALERTER_IMPL)
    pgFaults.fill(0, 10);
    QFile vmstat("/proc/vmstat"); //kernel 2.6+
    if (vmstat.exists() && vmstat.open( IO_ReadOnly )) {
        //qDebug("Memory Monitor: Linux 2.6");
        VMStatType = VMLinux_2_6;
        vmstat.close();
    } else {
        VMStatType = VMLinux_2_4;
        //qDebug("Memory Monitor: Linux 2.4");
   }

    if (VMStatType != VMUnknown) {
        vmMonitor = new QTimer( this );
        connect( vmMonitor, SIGNAL(timeout()), this, SLOT(memoryMonitor()));
        int freeMem = -1;
        readMemInfo(&freeMem);
        if (freeMem > MIN_MEM_LIMIT) {
            //有足够的内存，减少内存检查次数
            vmMonitor->start( 10000 );
            slowVMMonitor = TRUE;
        }
        else {
            vmMonitor->start( 1000 );
            slowVMMonitor = FALSE;
        }
    }
#endif

    //检查可用的存储空间
    Config cfg("qpe");
    cfg.setGroup("StorageMonitor");
    if (cfg.readBoolEntry("Enabled", TRUE)) {
        //qDebug("Storage Monitor enabled");
        sm = new StorageMonitor( this );
        QTimer *storageTimer = new QTimer( this );
        connect( storageTimer, SIGNAL(timeout()), sm, SLOT(checkAvailStorage()));
        storageTimer->start(cfg.readNumEntry("UpdatePeriod", 15)*1000);
    }

    ps = new PowerStatus; //这个类保存电池状态
    pa = new DesktopPowerAlerter( 0 );  //创建桌面电力报警对话框
    //根据电池状态在信道“QPE/System”发消息设置背景灯、设置屏幕保护时间间隔
    applyLightSettings(ps);

    if ( PluginLoader::inSafeMode() )//安全模式
         QTimer::singleShot(500, this, SLOT(showSafeMode()) );
    QTimer::singleShot(20*1000, this, SLOT(clearSafeMode()) );//清除安全模式

    //给 qwsServer 设置键盘过滤
    KeyFilter* kf = new KeyFilter(this);
    //将键盘信号与操作函数连接起来或转发信号
    connect( kf, SIGNAL(launch()), this, SIGNAL(launch()) );
    ……
    connect( kf, SIGNAL(backlight()), this, SLOT(toggleLight()) );

    connect( this, SIGNAL(volumeChanged(bool)), this, SLOT(rereadVolumes()) );
    rereadVolumes();
```

```
    serverApp = this;
}
```

6.2.3 Server 类构造函数

Server 类即窗口服务器，其构造函数不仅创建主窗口界面对象——Launcher 对象，并创建屏保对象、文档视图、包处理对象及应用程序启动对象等。它还构造用于 Qtopia Desktop 与设备通信的传输服务器、创建各种信道并连接各种系统消息。

Server 类的构造函数分析如下：

```
Server::Server() :
    QWidget( 0, 0, WStyle_Tool | WStyle_Customize ),
    qcopBridge( 0 ),
    transferServer( 0 ),
    packageHandler( 0 ),
    syncDialog( 0 ),
    soundserver( 0 ),
    qrr( 0 ),
    lastStartedApp( 0 )

    //设置内建命令，builtins 是静态的全局的 Command 结构变量数组，
    //这个数组中有诸如 Global::shutdown 的命令
    Global::setBuiltinCommands(builtins)      ;

    tid_xfer = 0;

    tsmMonitor = new TempScreenSaverMonitor(); //  用于屏幕保护的对象
    connect( tsmMonitor, SIGNAL(forceSuspend()), qApp, SIGNAL(power()) );// 屏幕挂起

#ifdef QTOPIA_PHONE
    serverGui = new PhoneImpl;  //用于手机的界面
#else
    serverGui = new Launcher;
#endif
    serverGui->createGUI();  // 创建窗口界面

    docList = new DocumentList( serverGui );  //     创建文档视图
    appLauncher = new AppLauncher(this);  //  用于启动应用程序的对象
    /*应用程序的状态有： enum ApplicationState { Launching, Running, Terminated, Raised }*/
    //在任务栏上设置应用程序运行状态条状态为 Raised
    connect(appLauncher, SIGNAL(raised(const QString&)), this
                         SLOT(applicationRaised(const QString&)) );
    //设置应用程序运行状态条为 Launching，限制应用程序运用的个数为 6，发信号关闭多余的应用程序
    connect(appLauncher, SIGNAL(launched(int      const QString&)), this
                        SLOT(applicationLaunched(int  const QString&)) );
    //从任务栏上取消应用程序运行状态条，关闭应用程序
    connect(appLauncher, SIGNAL(terminated(int,const QString&)), this,
                         SLOT(applicationTerminated(int,const QString&)) ) ;
    //设置任务栏上的应用程序运行状态条为 Running 正在运行状态
    connect(appLauncher, SIGNAL(connected(const QString&)), this
                         SLOT(applicationConnected(const QString&)) )    ;

    storage = new StorageInfo( this ); //用于得到闪存存储空间信息
    // 当存储空间发生变化时,如 SD 卡插拔时，更新窗口中应用程序及文档图标，更新符号连接等
    connect( storage, SIGNAL(disksChanged()), this, SLOT(storageChanged()) );

#ifdef QPE_HAVE_DIRECT_ACCESS
    QCopChannel *desktopChannel = new QCopChannel( "QPE/Desktop", this );
    connect( desktopChannel, SIGNAL(received( const QCString &, const QByteArray & )),
          this, SLOT(desktopMessage( const QCString &, const QByteArray & )) );
#endif
```

```
    //给系统 5秒钟启动
    qssTimerId = startTimer(5000);

    // 启动 services
    startTransferServer();//创建 QCopBridge 及 TransferServer 对象，端口 号分别为 4243和 4242
    (void) new IrServer( this );  //创建红外服务器对象

    packageHandler = new PackageHandler( this );  //创建包处理对象
    //发送设备按钮相关的服务请求消息
    connect(qApp, SIGNAL(activate(const DeviceButton*,bool)),this
                          SLOT(activate(const DeviceButton*,bool)));

    setGeometry( -10, -10, 9, 9 )     ;

    QCopChannel *channel = new QCopChannel("QPE/System", this);
    /*处理 Qtopia Desktop 到设备的系统消息，如设置安全授权、设置屏保模式、文档链接改变、创建目录、改变签名、开始同步、同步
结束、设置鼠标协议及设置键盘等*/
    connect(channel, SIGNAL(received(const QCString&,const QByteArray&))
                     this, SLOT(systemMsg(const QCString&,const QByteArray&)) );
    //重新装载 AppLnk 对象到 appLnkSet 中，声音报警并设置 LED 灯状态
    QCopChannel *tbChannel = new QCopChannel( "QPE/TaskBar", this );
    connect( tbChannel, SIGNAL(received(const QCString&,const QByteArray&))
        this, SLOT(receiveTaskBar(const QCString&,const QByteArray&)) );
    //删除 transferServer 和 qcopBridge
    connect( qApp, SIGNAL(prepareForRestart()), this, SLOT(terminateServers()) );
    //通知所有的 TimeMonitors 改变时区
    connect( qApp, SIGNAL(timeChanged()), this, SLOT(pokeTimeMonitors()) );
    //读出配置文件 Launcher 中 Preload组的 apps 的值，激活预装的应用程序
    preloadApps();
}
```

6.2.4 创建服务器主窗口界面 UI

服务器主窗口界面 UI 通过调用函数 createGUI()创建，它由窗口部件类 TaskBar 和 LauncherTabWidget构成，函数 createGUI()的调用层次如图 6-7 所示。

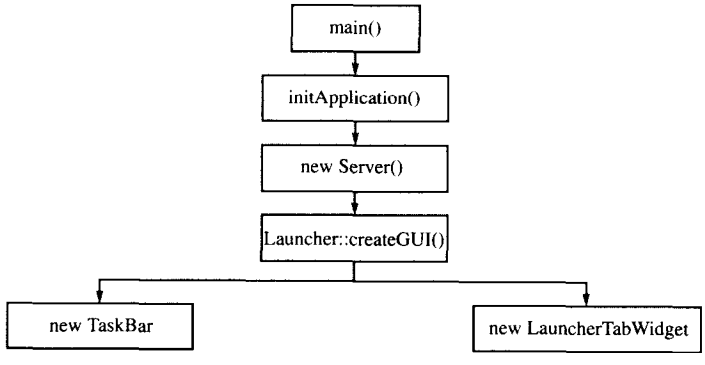


图 6-7 函数 createGUI()的调用层次

函数 createGUI()用于创建 PDA 主窗口图形界面，列出如下：

```
void Launcher::createGUI()
{
    setCaption( tr("Launcher") ) ;
    //用桌面大小设置窗口尺寸
    setGeometry( 0, 0, qApp->desktop()->width(), qApp->desktop()->height() );
    tb = new TaskBar;  //创建任务栏
    tabs = new LauncherTabWidget( this );  //   创建 TabBar
```

```
    setCentralWidget( tabs );

    ServerInterface::dockWidget( tb, ServerInterface::Bottom );
    qApp->installEventFilter( this );  //安装事件过滤
    //检查密码
    connect( qApp, SIGNAL(authenticate(bool)), this, SLOT(askForPin(bool)) );
    //显示选择的 Tab
    connect( tb,SIGNAL(tabSelected(const QString&)),this, SLOT(showTab(const QString&)) );
    //显示标题名字
    connect( tabs,SIGNAL(selected(const QString&)),this, SLOT(viewSelected(const QString&)) );
    //发消息运行视图中选中图标对应的 AppLnk 对象
    connect( tabs,SIGNAL(clicked(const AppLnk*)),this, SLOT(select(const AppLnk*)));
    //显示视图中选中图标对应的属性对话框
    connect( tabs,SIGNAL(rightPressed(const AppLnk*)),this,SLOT(properties(const AppLnk*)));

#if defined(Q_WS_QWS) && !defined(QT_NO_COP)
    QCopChannel* sysChannel = new QCopChannel( "QPE/System", this );
    connect( sysChannel, SIGNAL(received(const QCString&,const QByteArray&)),
           this, SLOT(systemMessage(const QCString&,const QByteArray&)) );
#endif

    //文档视图
    QImage img( Resource::loadImage( "DocsIcon" ) );
    QPixmap pm;
    pm = img.smoothScale( AppLnk::smallIconSize(), AppLnk::smallIconSize() );
    // 创建文档视图
    tabs->newView("Documents", pm, tr("Documents") )->setToolsEnabled( TRUE );
    QTimer::singleShot( 0, tabs, SLOT( initLayout() ) );//显示文档视图并设置 Tab 条

    qApp->setMainWidget( this );//将 Launcher 窗口设置为主窗口
}
```

(1) 任务栏类 TaskBar

构造函数 TaskBar()构造任务栏上的各种窗口部件，并连接处理 QPE/TaskBar信道消息，分析如下：

```
TaskBar::TaskBar() : QHBox(0, 0, WStyle_Customize | WStyle_Tool | WStyle_StaysOnTop | WGroupLeader)
{
    BackgroundMode mode = PaletteButton;
    setBackgroundMode(mode);//设置背景模式

    sm = new StartMenu( this );  //创建开始菜单 StartMenu 对象
    sm->setBackgroundMode(mode);
    connect(sm,SIGNAL(tabSelected(const QString&)),
                                this,SIGNAL(tabSelected(const QString&)) );

    inputMethods = new InputMethods( this ); //输入法
    inputMethods->setBackgroundMode(mode);
    connect( inputMethods, SIGNAL(inputToggled(bool)),this, SLOT(calcMaxWindowRect()) );

    stack = new QWidgetStack( this );
    stack->setBackgroundMode(mode);
    stack->setSizePolicy( QSizePolicy( QSizePolicy::Expanding, QSizePolicy::Minimum ) );
    label = new QLabel(stack);
    label->setBackgroundMode(mode);

    runningAppBar = new RunningAppBar(stack);  //运行状态栏
    runningAppBar->setBackgroundMode(mode);
    stack->raiseWidget(runningAppBar);

    waitIcon = new Wait( this );  //应用程序等运行时等待图标，一般为沙漏
    waitIcon->setBackgroundMode(mode);
    AppIcons *appIcons = new AppIcons( this );
    appIcons->setBackgroundMode(mode);
```

```
    sysTray = new SysTray( this );  //系统托盘，用来放置 applet 小应用程序图标
    sysTray->setBackgroundMode(mode);

    if (PluginLoader::inSafeMode()) {
        SafeMode *safeMode = new SafeMode( this );//用于显示安全模式的窗口部件
        safeMode->setBackgroundMode(mode);
    }
  ......
#if defined(Q_WS_QWS)
#if !defined(QT_NO_COP)
    QCopChannel *channel = new QCopChannel( "QPE/TaskBar", this );
    //处理 QPE/TaskBar 信道消息，如设置状态条文本、隐藏/显示输入法及托盘重装载 applets
    connect( channel, SIGNAL(received(const QCString&,const QByteArray&)),
                      this, SLOT(receive(const QCString&,const QByteArray&)) );
#endif
#endif
    //等待图标运行一段时间后应消失
    waitTimer = new QTimer( this );
    connect( waitTimer, SIGNAL( timeout() ), this, SLOT( stopWait() ) );
    clearer = new QTimer( this );
    //升起 runningAppBar，显示系统托盘 sysTray
    QObject::connect(clearer, SIGNAL(timeout()), SLOT(clearStatusBar()));
    //输入字符大小写切换
    connect( qApp, SIGNAL(symbol()), this, SLOT(toggleSymbolInput()) );
    //切换键盘上 NumLock 键
    connect( qApp, SIGNAL(numLockStateToggle()), this, SLOT(toggleNumLockState()) );
    //切换键盘上 CapsLock 键
    connect( qApp, SIGNAL(capsLockStateToggle()), this, SLOT(toggleCapsLockState()) );
}
```

6.2.5 创建应用程序图标视图

应用程序图标视图的创建是通过函数 reloadAppLnks() 完成的，这个函数在 Server 类构造函数创建对象时，被调用来创建图标及视图。函数 reloadAppLnks() 的调用层次如图 6-8 所示。

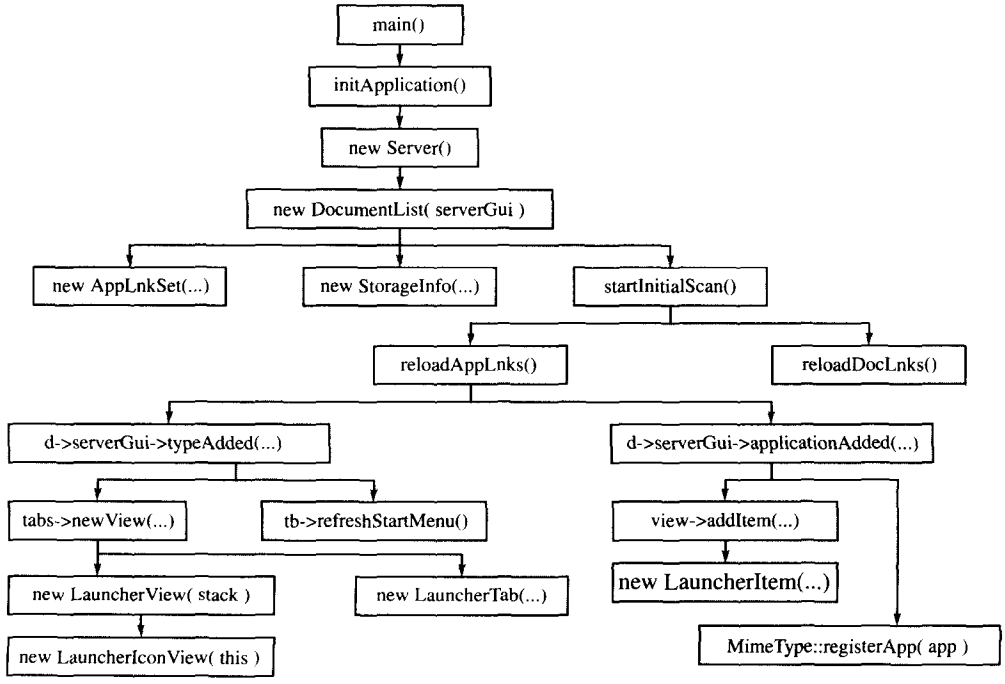


图 6-8　函数 reloadAppLnks()的调用层次

下面按照函数 reloadAppLnks() 调用的次序来分析这个函数。

函数 reloadAppLnks() 加载各种应用程序类型视图及 Tab 条到设备屏幕主窗口中，并将所有应用程序（包括游戏等）以图标方式加载到应用程序图标视图中。每个应用程序类型用一个.directory 文件描述，*.directory 说明了每个应用程序类型的名字及图标等。每个应用程序用一个.desktop 文件描述，其中描述在视图中应用程序使用的名字、图标，以及平台特征需求等。函数 reloadAppLnks() 读出这两个文件的内容，为每个应用程序建立对应的 AppLnk 对象，并把所有的 AppLnk 对象放入到 AppLnkSet 对象中。这样，应用程序即可通过这两个对象知道各个应用程序是如何在应用程序图标视图中显示的。并且从应用程序图标视图中的图标即可从 AppLnk 对象中查到对应的应用程序，从而可以运行这个应用程序。

函数 reloadAppLnks() 分析如下：

```
void DocumentList::reloadAppLnks()
{
   if ( d->sendAppLnks && d->serverGui ) {
        d->serverGui->applicationScanningProgress( 0 );  //进度条设置到0
        d->serverGui->allApplicationsRemoved();//将各个视图中的所有条目删除
   }

delete appLnkSet;
   /*将$QTOPIA_PATH/apps 目录下，将各个子目录下的 *.diretory 文件中的 [Desktop Entry] 组的 Name 项的值作为应用程序类型（如 Application、Game 及 Setting 等），将类型存入到这个子目录下的各个 appLnk 对象的成员 mType中，然后读出 Icon项值并找到对应的图标文件将各子目录各个*.desktop文件读到各个 appLnk 对象中，并将 appLnk 对象加到 AppLnkSet 对象中*/
    appLnkSet = new AppLnkSet( MimeType::appsFolderName() )  ;

   if ( d->sendAppLnks && d->serverGui ) {
      static QStringList prevTypeList;
      QStringList types = appLnkSet->types(); //得到应用程序类型存储的字符串链表
      for ( QStringList::Iterator ittypes=types.begin(); ittypes!=types.end(); ++ittypes) {
        if ( !(*ittypes).isEmpty() ) {
            if ( !prevTypeList.contains(*ittypes) ) {
               QString name = appLnkSet->typeName(*ittypes);  //得到应用程序类型名
               QPixmap pm = appLnkSet->typePixmap(*ittypes);//    每个应用类型的小图标
               QPixmap bgPm = appLnkSet->typeBigPixmap(*ittypes);//每个应用程序类型大图标

               if (pm.isNull()) {//如果小图标不够大，装入 UnknownDocument 图标
                    QImage img( Resource::loadImage( "UnknownDocument" ) );
                   pm = img.smoothScale( AppLnk::smallIconSize(), AppLnk::smallIconSize() );
                    bgPm = img.smoothScale( AppLnk::bigIconSize(), AppLnk::bigIconSize() )      ;
               }

                 //为每个应用程序类型创建新的视图并更新“开始”菜单
              d->serverGui->typeAdded( *ittypes, name.isNull() ? (*ittypes) : name, pm, bgPm );
            }
            prevTypeList.remove(*ittypes) ;
         }
        }
      //从 prevTypeList 链表中删除所有的应用程序类型
       for ( QStringList::Iterator ittypes=prevTypeList.begin(); ittypes!=prevTypeList.end(); ++ittypes)
{
          //qDebug("removing type %s", (*ittypes).latin1());
          d->serverGui->typeRemoved(*ittypes);
       }
       prevTypeList = types;//指向当前的应用程序类型字符串链表
   }

   QListIterator<AppLnk> itapp( appLnkSet->children() );
   AppLnk* l;
   //遍历 appLnkSet链表中所有的 AppLnk 对象
   while ( (l=itapp.current()) ) {
   ++itapp;
```

```
    if ( d->sendAppLnks && d->serverGui ) {
        //如果 l 所需要的系统特征满足，则添加应用程序到视图
        bool reqsSatisfied = TRUE;
        //读出 l 对象对应的.desktop 文件中的 Requires 项的值，req 是以为分隔符的多字符串
        QString req = l->property("Requires");

        if (!req.isEmpty()) {
            //以,为分隔符将多字符串分开，分开后分别放入到字符串链表中
            QStringList appReqs = QStringList::split(',', req);
            QStringList::ConstIterator it;
            for (it = appReqs.begin(); it != appReqs.end(); ++it) {
                /*如果全局的 QStringList *ServerApplication::features 中没有 *it 项,即系统中不含有要求的特征,条件
不满足*/
                if (!ServerApplication::haveFeature(*it)) {
                    reqsSatisfied = FALSE;  //系统中不含有需求的特征
                    break;
                }
            }
        }

        if (reqsSatisfied)
            d->serverGui->applicationAdded( l->type(), *l ); //添加应用程序到视图中
        }
    }

    if ( d->sendAppLnks && d->serverGui )
        d->serverGui->applicationScanningProgress( 100 );  //进度条进度到 100
}

// 这些是来自服务器的更新函数
void Launcher::typeAdded( const QString& type, const QString& name
                  const QPixmap& pixmap, const QPixmap& )
{
    tabs->newView( type, pixmap, name );  //加入新的视图及 Tab 条
    ids.append( type );  //将应用程序类型 type 放入到类内部的字符串链表 ids 中
    tb->refreshStartMenu();//更新开始菜单

    static bool first = TRUE; //  保证只显示一次
    if ( first ) {
    first = FALSE;
       tabs->categoryBar->showTab(type);  //显示 Tab
    }

    tabs->view( type )->setUpdatesEnabled( FALSE ) ;
    tabs->view( type )->setSortEnabled( FALSE ) ;
}
```

函数 newView 创建了应用程序图标视图及 Tab 条，分析如下：

```
LauncherView* LauncherTabWidget::newView( const QString& id, const QPixmap& pm
                                          const QString& label )
{
    LauncherView* view = new LauncherView( stack ) ;
    connect( view, SIGNAL(clicked(const AppLnk*)), this, SIGNAL(clicked(const AppLnk*)));
    connect( view, SIGNAL(rightPressed(const AppLnk*)),
                        this,SIGNAL(rightPressed(const AppLnk*))) ;

    int n = categoryBar->count();
    stack->addWidget( view, n ); //将 view 加入到 QWidgetStack 对象 stack 中
    //创建 Tab 条，并绘制 Tab 条上特殊的窗口、图标和文本
    LauncherTab *tab = new LauncherTab( id, view, pm, label ) ;
    categoryBar->insertTab( tab, n-1 );

    if ( id == "Documents" )  docview = view;
```

```
    Config cfg("Launcher");
    setTabAppearance( tab, cfg );  //设置 Tab 条外观，如字体、背景及前景颜色等
    return view;
}
```

函数 LauncherView 创建应用程序图标视图窗口并连接鼠标信号，分析如下：

```
LauncherView::LauncherView( QWidget* parent, const char* name, WFlags fl )
    : QVBox( parent, name, fl )
{
    catmb = 0;
    icons = new LauncherIconView( this );  //从 QIconView继承
    setFocusProxy(icons);
    /*在 icons->viewport()窗口部件中，设置鼠标操作方式，左键长时间按住相当于 PC 机下的鼠标右键模式*/
    QPEApplication::setStylusOperation( icons->viewport(), QPEApplication::RightOnHold )   ;
    //对视图进行设置
    icons->setItemsMovable( FALSE );
    icons->setAutoArrange( TRUE );
    icons->setSorting( TRUE );
    icons->setFrameStyle( QFrame::NoFrame );
    icons->setMargin( 0 )  ;
    icons->setSelectionMode( QIconView::NoSelection )  ;
    icons->setBackgroundMode( PaletteBase );
    icons->setResizeMode( QIconView::Fixed );
    vmode = (ViewMode)-1;
    setViewMode( Icon );

    //鼠标点击事件处理，得到点击图标的 AppLnk对象，并发出 clicked( appLnk )信号
    connect( icons, SIGNAL (mouseButtonClicked(int,QIconViewItem*,const QPoint&)),
             SLOT(itemClicked(int,QIconViewItem*)) );
    //图标选中状态发生变化，得到新选中图标的 AppLnk对象，并发出 clicked( appLnk )信号
    connect( icons, SIGNAL(selectionChanged()),
            SLOT(selectionChanged()) );
    //按了键盘回车键或相当于这个键的操作，发出 clicked( appLnk )信号
    connect( icons, SIGNAL(returnPressed(QIconViewItem*)),
            SLOT(returnPressed(QIconViewItem*)) );
    //用鼠标按住应用程序图标，得到选中图标的 AppLnk对象，发出 rightPressed( appLnk )信号
    connect( icons, SIGNAL(mouseButtonPressed(int,QIconViewItem*,const QPoint&)),
            SLOT(itemPressed(int,QIconViewItem*)) );

    tools = 0;
    //Ruled是定义的背景枚举类型，设置背景图案及颜色
    setBackgroundType( Ruled, QString::null );
}
```

函数 applicationAdded 创建应用程序图标对应的 LaunchItem 对象放在视图中，并注册应用程序对应的 MIME 类型，列出如下：

```
void Launcher::applicationAdded( const QString& type, const AppLnk& app )
{
    if ( app.type() == "Separator" ) // No tr
    return;

    LauncherView *view = tabs->view( type );
    if ( view )
    view->addItem( new AppLnk( app ), FALSE ); //创建新的 LaunchItem对象放在视图中
    else
    qWarning("addAppLnk: No view for type %s. Can't add app %s!"
                    type.latin1(),app.name().latin1() );

    MimeType::registerApp( app ); //    注册应用程序对应的 MIME 类型
}
```

6.3 应用程序启动过程

在 LauncherView 视图窗口中，当用户点击应用程序图标时会产生 mouseButtonClicked 信号，这个信号与 LauncherView 类中的槽函数 itemClicked(int btn, QIconViewItem *item) 连接起来。函数 itemClicked 得到 *item 对应的 AppLnk 对象 appLnk，并发出信号 clicked（appLnk）。用户点击应用程序图标时的函数调用层次如图 6-9 所示。

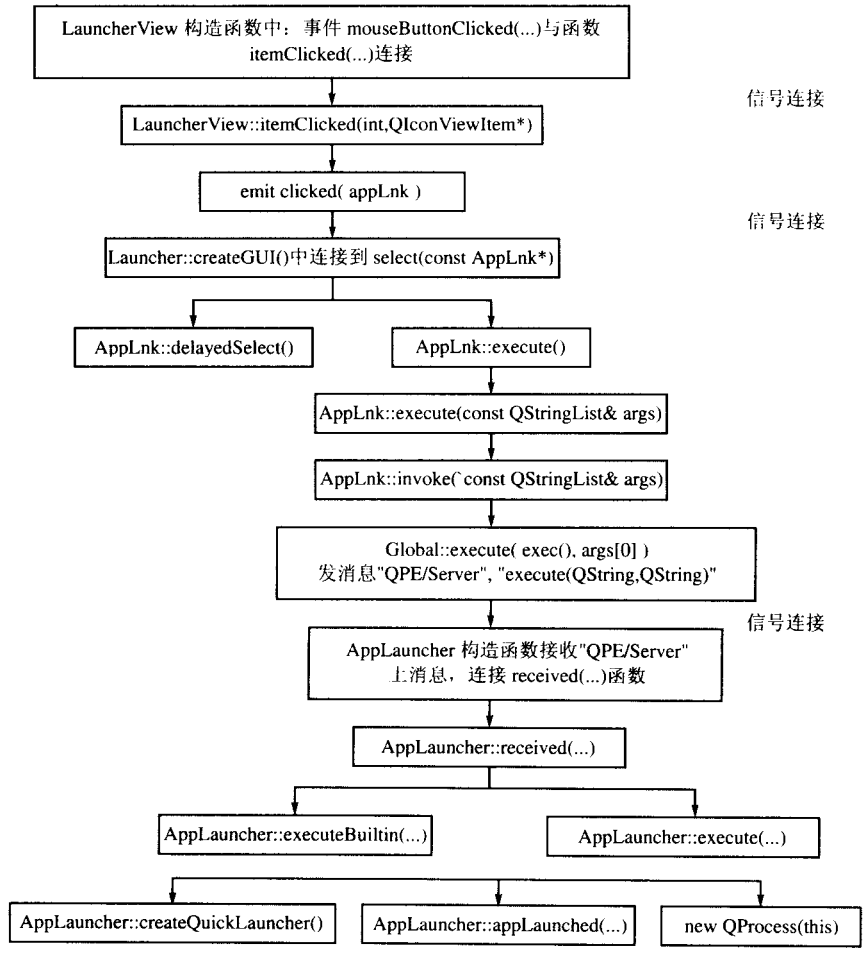


图 6-9 用户点击应用程序图标时的函数调用层次

LauncherView 类中与鼠标点击事件相关的代码列出如下：

```
LauncherView::LauncherView( QWidget* parent, const char* name, WFlags fl )
    : QVBox( parent, name, fl )
{
    ......
    //鼠标点击事件信号与槽函数 itemClicked 连接
    connect( icons, SIGNAL(mouseButtonClicked(int,QIconViewItem*,const QPoint&)),
             SLOT(itemClicked(int,QIconViewItem*)) );
    ......
}
```

```
void LauncherView::itemClicked( int btn, QIconViewItem *item )
{
    if ( item ) {
        AppLnk *appLnk = ((LauncherItem *)item)->appLnk();
        if ( btn == LeftButton ) {//是鼠标左键
            QTOPIA_PROFILE("icon clicked");
            // 确保将运行的图标条目是高亮选中状态
            icons->setCurrentItem( item );
            emit clicked( appLnk );
        }
        item->setSelected(FALSE);
    }
}
```

在函数 createGUI()中有信号 clicked(const AppLnk*)响应的函数 select(const AppLnk*),信号连接的代码列出如下：

```
void Launcher::createGUI()
{
    ......
    connect( tabs, SIGNAL(clicked(const AppLnk*)),
                   this, SLOT(select(const AppLnk*)));
    ......
}

void Launcher::select( const AppLnk *appLnk )
{
    if ( appLnk->type() == "Folder" ) { // No tr
    // 不支持文件夹
    } else {
        if ( appLnk->exec().isNull() ) {//应用程序不存在
            if ( ! delayedAppLnk ) {
                delayedAppLnk = new AppLnk(*appLnk);
              //给出错误提示信息
                QTimer::singleShot( 0, this, SLOT(delayedSelect()) );
            }
        } else {//当应用程序存在时
            tabs->setBusy(TRUE);
            emit executing( appLnk );
         appLnk->execute();  //发信号运行应用程序
        }
    }
}
```

当应用程序图标对应的应用程序不存在时，函数 delayedSelect()给出错误信息提示窗口，并根据用户的选择来决定是否用文本应用程序来打开，该函数列出如下：

```
void Launcher::delayedSelect()
{
    if ( !delayedAppLnk )
    return;
    AppLnk *appLnk = delayedAppLnk;
    delayedAppLnk = 0;
    //给出错误提示信息对话框
    int i = QMessageBox::information(this,tr("No application"),
         tr("<qt>No application is defined for this document. "
         "Type is %1.</qt>").arg(appLnk->type()),
         tr("OK"), tr("View as text"), 0, 0, 1);

    if ( ! QFileInfo(appLnk->file()).exists() )
        qDebug( "file no longer exists!" );
    else if ( i == 1 )  //用户选择OK按钮，发信号使用文本应用程序打开文件
        Global::execute(Service::app("Open/text/*"),appLnk->file());
```

```
    delete appLnk;
}
```

函数 execute()运行 AppLnk 对象相关的应用程序，列出如下：

```
void AppLnk::execute() const
{
    execute(QStringList());
}
```

函数 execute 处理 display 的旋转并触发与这个 AppLnk 对象相关的应用程序，参数 args 是应用程序的参数。该函数分析如下：

```
void AppLnk::execute(const QStringList& args) const
{
#ifdef Q_WS_QWS
    if ( !mRotation.isEmpty() ) {  // mRotation 存储的旋转度数字符串不为空
        // 下面仅在 Qtopia 的 server 中起作用
        //从$HOME/Settings/qpe.conf 文件中读取 Rotation 组 Screen 项的值，得到旋转度数
        int rot = QPEApplication::defaultRotation();
        rot = (rot+mRotation.toInt())%360;
        QCString old = getenv("QWS_DISPLAY");

       //为当前的 AppLnk 对象保存当前的 display id
       int displayID = 0;
       QRegExp regExp( ":[0-9]+" );  //支持超过 10 个 display
       int length;
       int match = regExp.match( old , 0, &length );
       if ( match >= 0 ) {//如果 old 字符串匹配":[0-9]+", match 为匹配位置
           QString num = QString(old).mid( match+1, length-1 );
           displayID = num.toInt();
       }

        setenv("QWS_DISPLAY", QString("Transformed:Rot%1:%2")
            .arg(rot).arg(displayID), 1); //设置旋转的 display
        invoke(args);
        setenv("QWS_DISPLAY", old.data(), 1);  //恢复当前 display
    } else
        invoke(args);//触发与这个 AppLnk 对象相关的应用程序
#endif
}
```

函数 invoke 触发与这个 AppLnk 对象相关的应用程序，该函数没有考虑窗口的旋转问题，因而不能直接被调用，列出如下；

```
void  AppLnk::invoke(const QStringList& args)
{
    Global::execute( exec(), args[0] );  //exec()返回可执行的应用程序名字，args[0]为文档名
}
```

函数 execute 发送消息运行参数 c 标识的应用程序，如果应用程序 c 不为空，则传递参数 document。注意通过应用程序的 QPE/Application/{appname}信道发送一个 QCop消息才是调用这个函数的好方法，该函数分析如下：

```
void Global::execute( const QString &c, const QString& document )
{
#ifndef QT_NO_COP
    if ( document.isNull() ) {
    QCopEnvelope e( "QPE/Server", "execute(QString)" );
    e << c;
```

```
    } else {
    QCopEnvelope e( "QPE/Server", "execute(QString,QString)" );
    e << c << document;
    }
#endif
    return;

}
```

应用程序的执行在 AppLauncher 类中完成，该类的构造函数中与 QPE/Server 消息连接的代码列出如下：

```
AppLauncher::AppLauncher(QObject *parent, const char *name)
    : QObject(parent, name), qlProc(0), qlPid(0), qlReady(FALSE),
      appKillerBox(0)
{
   ......
    channel = new QCopChannel( "QPE/Server", this );
    connect( channel, SIGNAL(received(const QCString&,const QByteArray&)),
         this, SLOT(received(const QCString&,const QByteArray&)) );
    ......
}
```

函数 received 是处理 QPE/Server 消息的槽函数，主要用来处理应用程序的运行，列出如下：

```
void AppLauncher::received(const QCString& msg, const QByteArray& data)
{
    QDataStream stream( data, IO_ReadOnly );
    if ( msg == "execute(QString)" ) {
        QString t;
        stream >> t;
        if ( !executeBuiltin( t, QString::null ) )
            execute(t, QString::null);
    } else if ( msg == "execute(QString,QString)" ) {
       QString t,d;
       stream >> t >> d;
       if ( !executeBuiltin( t, d ) )
           execute( t, d );
    } else if ( msg == "processQCop(QString)" ) {
       //消息排队
       if (mDelayMessages) {
          queueMessage(msg, data); //消息在内部变量 qmessages 中排队
          return;
       }
       QString t;
       stream >> t;
       if ( !executeBuiltin( t, QString::null ) )
           execute( t, QString::null, TRUE);
    } else if ( msg == "raise(QString)" ) {
      QString appName;
      stream >> appName;
    ......
    } else if ( msg == "sendRunningApps()" ) {  //向 Qtopia Desktop 发送运行的应用程序的名字
       QStringList apps;
       QMap<int,RunningApp>::Iterator it;
       for( it = runningApps.begin(); it != runningApps.end(); ++it )
           apps.append( (*it).name );
        QCopEnvelope e( "QPE/Desktop", "runningApps(QStringList)" );
        e << apps;
    } else if ( msg == "appRaised(QString)" ) {
       QString appName;
       stream >> appName;
       qDebug("Got a heartbeat from %s", appName.latin1());
```

```
        QMap<QString,int>::Iterator it = waitingHeartbeat.find(appName);
        if ( it != waitingHeartbeat.end() ) {
            killTimer( *it );
            waitingHeartbeat.remove(it);
        }
        // 检查确信我们没有正在等待在用户输入
        if ( appKillerBox && appName == appKillerName ) {
            //如果我们正在等待用户输入，杀死对话框
            delete appKillerBox;
            appKillerBox = 0;
            appKillerName = QString::null;
        }
        runningList.remove(appName);
        runningList.prepend(appName);  //加入运行的应用程序的名字
        emit raised(appName);
    }
}
```

如果运行的应用程序名为 Qtopia 内建命令，函数 executeBuiltin 则执行内建命令并返回 TRUE；否则返回 FALSE，该函数分析如下：

```
bool AppLauncher::executeBuiltin(const QString &c, const QString &document)
{
    //得到内建命令，内建命令存在静态变量 Command* builtin 上
    Global::Command* builtin = Global::builtinCommands();
    QGuardedPtr<QWidget> *running = Global::builtinRunning();  //正在运行的窗口

    // 尝试使用 builtin 执行这个应用程序
    if (builtin) {
        for (int i = 0; builtin[i].file; i++) { //遍历内建命令的文件名
            if ( builtin[i].file == c ) {  //如果内建命令为需要执行的应用程序 c
                if ( running[i] ) {//如果在运行队列中，则升起这个命令的窗口
                    if ( !document.isNull() && builtin[i].documentary )
                        Global::setDocument(running[i], document);
                    running[i]->raise();
                    running[i]->show();
                    running[i]->setActiveWindow();
                } else {
                //如果不在运行队列中，运行这个内建命令，并加入窗口部件指针到运行队列
                    running[i] = builtin[i].func( builtin[i].maximized );
                }
#ifndef QT_NO_COP
                QCopEnvelope e("QPE/System", "notBusy(QString)" );
                e << c; // that was quick ;-)
#endif
                return TRUE;
            }
        }
    }

    // 转化命令行到一个参数链表
    QStringList list = QStringList::split(QRegExp("  *"),c);
    QString ap=list[0];  //得到应用程序名

    if ( ap == "suspend" ) { //如果应用程序名为“suspend”，它与按键 Key_F34 产生的功能一样
        QWSServer::processKeyEvent( 0xffff, Qt::Key_F34, FALSE, TRUE, FALSE );
        return TRUE;
    }
    return FALSE;
}
```

函数 execute 运行应用程序，其参数 c 是应用程序名及应用程序参数，参数 docparam 是应用程序启动的文档参数。当应用程序已运行时，改写应用程序的 *pid* 信道的内容。如果有文档，发送消息让

已运行的应用程序打开新的文档。如果应用程序是应用程序组成的库，则使用 Quick Launcher 应用程序加载库来运行应用程序窗口；如果应用程序是单独可执行的应用程序，则使用 QProcess 类来启动应用程序，QProcess 类最终是调用 C 库函数 execve 或函数 execvp 运行应用程序的。应用程序启动后，将应用程序名等放入全局变量 runningApps 和 runningList 中，以便系统对其进行管理。

函数 execute 分析如下：

```
bool AppLauncher::execute(const QString &c, const QString &docParam, bool noRaise)
{
    // 转换命令行到一个参数链表
    QStringList list = QStringList::split(QRegExp("  *"),c);
    if ( !docParam.isEmpty() )list.append( docParam );

    QString appName = list[0];  //得到应用程序名
    if ( isRunning(appName) ) {//如果应用程序正在运行
        QCString channel = "QPE/Application/";
        channel += appName.latin1();

        //需要锁住它来避免与QPEApplication::processQCopFile的竞争
        QFile f(Global::tempDir() + "qcop-msg-" + appName);
        if ( !noRaise && f.open(IO_WriteOnly | IO_Append) ) {
#ifndef Q_OS_WIN32
            flock(f.handle(), LOCK_EX);
#endif

            QDataStream ds(&f);
            QByteArray b;
            QDataStream bstream(b, IO_WriteOnly);
            if ( !f.size() ) {
                ds << channel << QCString("raise()") << b;  //写入ds
                if ( !waitingHeartbeat.contains( appName ) && appKillerName != appName ) {
                    int id = startTimer(RAISE_TIMEOUT_MS);
                    waitingHeartbeat.insert( appName, id );
                }
            }
            if ( !docParam.isEmpty() ) {//如果docParam不为空
                bstream << docParam;
                ds << channel << QCString("setDocument(QString)") << b; //写入ds
            }

            f.flush();
#ifndef Q_OS_WIN32
            flock(f.handle(), LOCK_UN);
#endif
            f.close();
        }
        if ( QCopChannel::isRegistered(channel) ) //避免不必要的警告
            QCopChannel::send(channel,"QPEProcessQCop()");

        return TRUE;
    }
    //将参数字符串从QString转换为char格式
    QStrList slist;
    unsigned j;
    for ( j = 0; j < list.count(); j++ )     slist.append( list[j].utf8() );//参数的字符串转换

    const char **args = new const char *[slist.count() + 1];
    for ( j = 0; j < slist.count(); j++ )    args[j] = slist.at(j);  //得到应用程序参数字符串
    args[j] = NULL;

    //使用Quick launcher启动插件或应用程序编译成的库（即不含main函数，只有窗口部件组成）
    if ( qlPid && qlReady
&& QFile::exists( QPEApplication::qpeDir()+"plugins/application/lib"+args[0] + ".so" )) {
```

```
        {//如果库存在，使用Quick launcher启动
            qDebug( "Quick launching: %s", args[0] );
            if ( getuid() == 0 )  setpriority( PRIO_PROCESS, qlPid, 0 );//设置运行优先级
            QCString qlch("QPE/QuickLauncher-");
            qlch += QString::number(qlPid);  //组成pid信道名
            QCopEnvelope env( qlch, "execute(QStrList)" );
            env << slist;
            QTOPIA_PROFILE("sending qcop to quicklauncher");  //打印信息
           /*将程序名args[0]及qlPro组成的结构RunningApp对象app放入全局变量runningApps[pid]中，将args[0]放入全局变量runningList中，向"QPE/System"信道发出"busy()"消息*/
            appLaunched(qlProc, args[0]);
            qlProc = 0;
            qlPid = 0;
            qlReady = FALSE;
           //启动运行$QTOPIA_PATH/bin/quicklauncher
            QTimer::singleShot( getuid() == 0 ? 800 : 1500, this, SLOT(createQuickLauncher()) );
           delete [] args;
            return TRUE;
        }
        //使用QProcess来启动应用程序
        QProcess *proc = new QProcess(this);
        proc->setCommunication(0);
        proc->addArgument(QPEApplication::qpeDir()+"bin/"+args[0]);  //加入应用程序名
        for (int i=1; i < (int)slist.count(); i++)    proc->addArgument(args[i]);
        connect(proc, SIGNAL(processExited()), this, SLOT(processExited()));
        bool started = proc->start();  //运行应用程序
        if (!started) {//如果没有启动
            proc->clearArguments();
            for (int i=0; i < (int)slist.count(); i++)   proc->addArgument(args[i]);
            started = proc->start();
        }

        if (!started) {
            qDebug("Unable to start application %s", args[0]);
            delete proc;
        } else {

            appLaunched(proc, args[0]);
        }
        delete [] args;
        return started;
    }
```

函数 appLaunched 将启动的应用程序放入到全局变量 runningApps 和 runningList 中，以便对这个运行的应用程序进行管理，列出如下：

```
void AppLauncher::appLaunched(QProcess *proc, const QString &appName)
{
    RunningApp app;
    app.name = appName;
    app.proc = proc;
    int pid = (int)proc->processIdentifier();
    runningApps[pid] = app;
    runningList.remove(appName);
    runningList.prepend(appName);
    emit launched(pid, appName);
    QCopEnvelope e("QPE/System", "busy()");
}
```

6.4 文档的打开过程

在 Qtopia 中，文件管理器是 filebrowser。它使用 FileView 类窗口部件作为视图窗口，使用 FileItem

类对象代表各个文件或文件夹条目显示在视图窗口中。当用户双击文件条目时，如果文件条目是应用程序，应用程序将通过调用函数 execve 启动运行；如果文件条目是文档，则将被 MIME 类型所支持的应用程序打开。

在 Qtopia 中，每个文档都有一个同名的.desktop 文件描述文档的运行配置，其中包括文档的 MIME 类型、归属的类别（即在哪一个 Tab 窗口中显示）、显示文件名和文件名等。当打开一个文档时，找到同名的.desktop 文件即可知道其 MIME 类型等，从而可找到对应的应用程序来打开文件。

FileView 类构造函数将鼠标双击事件信号与槽函数 itemDblClicked 连接起来，该构造函数中信号连接部分代码列出如下:

```
FileView::FileView( const QString & dir, QWidget * parent,
                    const char * name )
    : QListView( parent, name ),
      menuTimer( this ),
      le( NULL ),
      itemToRename( NULL )
{
    ......
    connect( this, SIGNAL( doubleClicked(QListViewItem*)),
             SLOT( itemDblClicked(QListViewItem*)) );
    ......
}
```

函数 itemDblClicked 处理鼠标双击事件，列出如下：

```
void FileView::itemDblClicked( QListViewItem * i)
{
    FileItem * t = (FileItem *) i;

    if(t == NULL) return;
    if(t->launch() == -1){
    QMessageBox::warning( this, tr( "Launch Application" ),
         tr( "Launch failed!" ), tr( "OK" ) );
    }
}
```

FileItem 类从 QListViewItem 继承，代表文件管理器窗口中显示的每个条目。从 FileItem 类可得到其对应条目的 QFileInfo 对象，每个文档或应用程序都有一个同名 .desktop文件来描述其各种配置信息。FileItem 类的成员函数 launch()用来运行条目，条目可能对应 3 种情况，即可执行应用程序、service 和文档。对于可执行应用程序，调用 execl()函数来运行它；对于 service，发消息 openFile(QString)打开它；对于文档，调用 AppLnk 类的 execute()函数打开它。

函数 launch()分析如下：

```
int FileItem::launch()
{
    // fileInfo 是在文件管理器窗口选中条目对应文件的 QFileInfo 对象
    //得到文件路径，如$HOME/Documents/text/plain/test.txt
    QString     file_to_launch = fileInfo.filePath();
    MimeType    mt(fileInfo.filePath());  //MIME 类型，如 text/plain
    QString     fname_desktop(file_to_launch);

    if (file_to_launch.contains(QDir::homeDirPath() + "/Documents")) {
         int    idx;
         //去掉文件名中的 '.' 后面的字符，它是扩展名标志
         idx = fname_desktop.findRev('.');
         if (idx != -1) {
            fname_desktop.truncate(idx);
         }
```

```
            //组成.desktop 文件名，如$HOME/Documents/text/plain/test.desktop
            fname_desktop += ".desktop";

            //由文档全路径名去掉路径得到不带路径的.desktop 文件名
             idx = fname_desktop.findRev('/');
             if (idx != -1) {
                fname_desktop = fname_desktop.right(fname_desktop.length() - idx);
             }
            //重组合成.desktop 全路径文件名，如$HOME/Documents/text/plain/test.desktop
             fname_desktop = QDir::homeDirPath() + "/Documents/" + mt.id() +fname_desktop;
        }

        if (!QFile::exists(fname_desktop)) {  //如果.desktop 文件不存在
            //我们尝试打开在$QPEDIR/Documents 下的.desktop 文件，如果不存在，则重打开最初请求的文件
              fname_desktop = fileInfo.filePath();
        }

        QString type = mt.id();  //MIME 类型，如 text/plain
        //如果是可执行文件且是 application/octet-stream 类型，则使用 execl 函数运行
        if (isExecutable() && type.contains("application/octet-stream") ) {
            int cpid = fork();

            switch (cpid){
               case -1:    //不能创建进程
               QMessageBox::warning( 0, QObject::tr( "File Manager" ),
                         QObject::tr( "<qt>Cannot fork this process.</qt>" ),
                         QObject::tr( "&OK" ) );
                  break;
               case 0:
                  qDebug("Filebrowser: executing " + file_to_launch);
                  execl(file_to_launch, file_to_launch, 0);
                  break;
               default:
                  //父进程
                  break;
            }
    } else {//如果不是可执行文件
       //打开$HOME/Settings/Service-open-<type>.conf 文件
          Config cfg(Service::appConfig("Open/"+type),Config::File);
          cfg.setGroup("Standard");  //设置组为 Standard
          if ( cfg.readNumEntry("Version") > 100 ) { //读取 Version 项的值，如果值大于 100，是 service
              // 用来打开 service
              /*从$HOME/Settings/Service-open-<type>.conf 文件的 service 组的 appname 项读出应用程序名，在应用程序名信
道上发送 openFile(QString)消息*/
             QCopEnvelope e(Service::channel("Open/"+type), "openFile(QString)");
             e << fname_desktop;
          } else {  //是文档
             //用来打开文档
             DocLnk doc( fname_desktop, FALSE ); //DocLnk 类从 AppLnk 类继承
             doc.execute();  //调用父类 AppLnk 的 execute()函数运行文档对应的应用程序
          }
       }

       listView()->clearSelection(); //清除选中状态
       return 1;
    }
```

6.5 池监控小应用程序

电池监控小应用程序是一个插件，它在托盘上显示电池的状态，以一个图标表示电池的电量。该应用程序使用 BatteryMeter 类在托盘上显示小图标，使用 BatteryStatus 以一个窗口形式显示电池状态，这两个类在屏幕上的位置如图 6-10 所示。

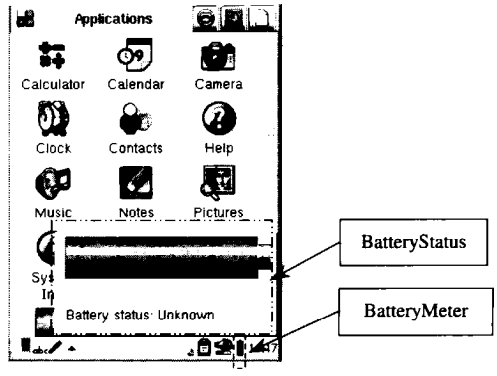


图 6-10　两个类在屏幕上的位置

电池监控小应用程序使用 BatteryAppletImpl 类作为插件的接口，使用 PowerStatusManager 类从 /proc/apm 文件中读到电源状态。读取电池状态后，电池监控小应用程序插件可以显示它，插件的装载是通过插件装载器 PluginLoader类来实现的。

6.5.1　电池状态窗口类

电池状态窗口类说明如下。

（1）BatteryMeter 类

BatteryMeter 类从 QWidget 类继承，用于保存 BatteryStatus 类和 PowerStatus 类对象指针。它从电源管理器读电池容量，根据容量绘制电池图标。该类的成员函数及其功能说明如表 6-18 所示。

表 6-18　BatteryMeter 类的成员函数及其功能说明

成员函数名	功能说明
BatteryMeter(QWidget *parent)	构造函数，创建 PowerStatus 对象、设置窗口部件的大小、创建定时器，并连接函数 chargeTimeout()
chargeTimeout()　[slot]	电池充电时，每隔一定时间根据电池容量绘制表示容量的图标
paintEvent(QPaintEvent*)	根据电池容量完成对图标绘制
sizeHint()	电池图标的尺寸
mousePressEvent(QMouseEvent *)	如果 BatteryStatus 窗口可见，关闭它；如果 BatteryStatus 窗口不可见，创建并显示它
timerEvent(QTimerEvent *)	每隔一定时间，通过电源管理器读出电池的容量，通过图标显示电池的容量。若非充电状态，则停止充电定时器

（2）BatteryStatus 类

BatteryStatus 类从 QFrame 类继承，用于保存 PowerStatus 对象指针。它根据电池电量在窗口部件中绘制电池图标及电池状态描述字符串。该类的成员函数及其功能说明如表 6-19所示。

表 6-19　BatteryStatus 类的成员函数及其功能说明

成员函数名	功能说明
BatteryStatus(...)	设置窗口部件背景颜色，得到电池电量的百分值，设置标题文字
sizeHint()	得到窗口部件的尺寸
updatePercent(int pc, bool doRepaint)	更新电池电量值的显示
drawSegment(QPainter *p,...)	根据电池电量绘制电池图标
drawContents(QPainter *p)	根据电池电量绘制电池图标及描述字符
statusText()	根据电池电量设置电池状态标签上的文本描述，并返回文本字符串

（3）BatteryAppletImpl 类

BatteryAppletImpl 类从 TaskbarAppletInterface 继承，用于保存 BatteryMeter 对象指针。它封装了 BatteryMeter 对象与外界的接口，通过函数 queryInterface 可以查找到 BatteryMeter 对象指针。该类的成员函数及其功能说明如表 6-20 所示。

表 6-20 BatteryAppletImp1 类的成员函数及其功能说明

成员函数名	功能说明
applet(QWidget *parent)	创建 BatteryMeter 对象，并返回这个对象指针
position()	返回窗口部件的位置序号值，这里为 8
queryInterface(const QUuid &uuid, QUnknownInterface **iface)	返回这个插件接口类对象的指针，引用计数加 1
Q_EXPORT_INTERFACE()	创建 BatteryMeter 对象，并返回这个对象指针

BatteryAppletImpl 类的父类 TaskbarAppletInterface 从 QUnknownInterface 类继承，它在通用插件接口的基础上增加了两个接口虚函数。TaskbarAppletInterface 类（在 src/libraries/qtopia/taskbarappletinterface.h 中）列出如下：

```
struct QTOPIA_PLUGIN_EXPORT TaskbarAppletInterface : public QUnknownInterface
{
    virtual QWidget *applet( QWidget *parent ) = 0;
    virtual int position() const = 0;
};
```

插件的通用接口 QUnknownInterface 定义了通用接口虚函数，使用宏定义 QTOPIA_PLUGIN_EXPORT 在目标文件中声明了结构 QUnknownInterface 的地址。这样，应用程序可以查询到结构 QUnknownInterface 的地址。结构 QUnknownInterface（在 src/libraries/qtopia/qcom.h 中）的声明代码列出如下：

```
#define QTOPIA_PLUGIN_EXPORT __declspec(dllexport)

struct QTOPIA_PLUGIN_EXPORT QUnknownInterface
{
    virtual QRESULT queryInterface( const QUuid&, QUnknownInterface** ) = 0;
    virtual ulong   addRef() = 0;
    virtual ulong   release() = 0;
};
```

BatteryAppletImpl 类重载了接口查询函数 queryInterface，用来查找编号为 uuid 的插件接口。并将接口指针放在 iface 中返回，使用 QUuid 是为了使接口编号是惟一存在的。函数 queryInterface 的实现列出如下：

```
QRESULT BatteryAppletImpl::queryInterface( const QUuid &uuid, QUnknownInterface **iface )
{
    *iface = 0;
    if ( uuid == IID_QUnknown )
        *iface = this;
    else if ( uuid == IID_TaskbarApplet )
        *iface = this;
    else
        return QS_FALSE;

    (*iface)->addRef();

    return QS_OK;
}
```

6.5.2 电源状态管理器类

电源状态管理器类说明如下。

（1）PowerStatus 类

PowerStatus 类以 PowerStatusManager 类作为友类，用于从 PowerStatusManager 类中得到电源的各种状态，并以 API 函数的方式返回状态值。

（2）PowerStatusManager 类

PowerStatusManager 类得到电源状态，用于保存静态的 PowerStatus 和 PowerStatusManager 类对象指针。通过读取文件/proc/apm，PowerStatusManager 类的成员函数 APMEnabled() 返回 APM 是否激活的值，函数:getProcApmStatus 得到电源的各种状态值。

6.5.3 插件装载

插件装载通过插件装载器 PluginLoader 类来完成，PluginLoader 类将某一类型插件的窗口部件指针及名字指针放在插件链表中返回，这样创建 PluginLoader 对象的应用程序即可调用插件的窗口部件。

任务栏上的托盘加入 applet 插件的函数 addApplets 分析如下：

```
void SysTray::addApplets()
{
    hide();
    delete loader;
    loader = new PluginLoader( "applets" );//      创建 applets 类型插件对象

    QStringList faulty;

    QStringList list = loader->list();//得到含有窗口部件指针及名字指针的链表
    TaskbarApplet **applets = new TaskbarApplet* [list.count()];//     创建对象数组
    QStringList::Iterator it;
    int napplets=0;
    for ( it = list.begin(); it != list.end(); ++it ) {
        TaskbarAppletInterface *iface = 0;

        //查询插件接口,得到窗口部件指针 iface
        if ( loader->queryInterface( *it, IID_TaskbarApplet,
                (QUnknownInterface**)&iface ) == QS_OK ) {
            TaskbarApplet *applet = new TaskbarApplet;
            applets[napplets++] = applet;
            applet->iface = iface;//存入窗口部件对象指针
            applet->name = *it;

        } else {//查询插件接口失败，截掉 name 中的.so 或 applet 部分

            //与任务栏设置使用的一样对 name 进行处理
            QString name = (*it).mid(3) ;
            int sep = name.find( ".so" );

            if ( sep > 0 )
                name.truncate( sep );
            sep = name.find( "applet" )   ;
            if ( sep == (int)name.length() - 6 )
                name.truncate( sep );
            name[0] = name[0].upper();//    第 1 个字母大写
            faulty += name;
            // 对有读取接口错误的插件不做任何事情
        }
```

```
    }

    //将插件链表中的插件按位置序号排序
    qsort(applets,napplets,sizeof(applets[0]),compareAppletPositions);

    //设置每个插件的背景模式，隐藏窗口宽度小于1的插件
    while (napplets--) {
        TaskbarApplet *applet = applets[napplets];
        applet->applet = applet->iface->applet( this );
        applet->applet->setBackgroundMode( PaletteButton );

        if ( applet->applet->maximumSize().width() <= 1 )
            applet->applet->hide();

        appletList.append(applet);
    }
    delete [] applets;

    show();  //显示托盘窗口部件
}
```

6.6 外观设置

Qtopia 外观设置是一个应用程序，它可以设置窗口系统的颜色、风格、字体及构架风格等，对应由颜色、风格、字体及构架风格等视图窗口部件组成。当用户点击每个视图中的条目时，从样例窗口中可预览到窗口外观改变后的效果。当点击 OK 按钮后，Qtopia 窗口系统的外观将改变到用户设置的外观样式。

Qtopia外观风格配置保存在$HOME/Settings/qpe.conf 文件中的 Appearance 组中，其内容列出如下：

```
[Appearance]
AlternateBase = #EEF5D6
Background = #eeeeee
BackgroundImage =
Base = #FFFFFF
Button = #F0F0F0
ButtonText = #000000
Decoration = Qtopia
FontFamily = Helvetica
FontSize = 12
Foreground = #222222
Foreground_disabled = #BBBBBB
Highlight = #8BAC30
HighlightedText = #FFFFFF
IconSize = 16
Scheme = Qtopia
Shadow =
Style = Qtopia
Text = #000000
Text_disabled = #BBBBBB
Theme = qtopia.conf
```

Qtopia 外观设置应用程序的主窗口部件由 AppearanceSettings 类描述，AppearanceSettings 类的构造函数运行完成后，主窗口界面创建完成。 Qtopia 外观设置应用程序运行后的窗口 UI 界面如图 6-11 所示。

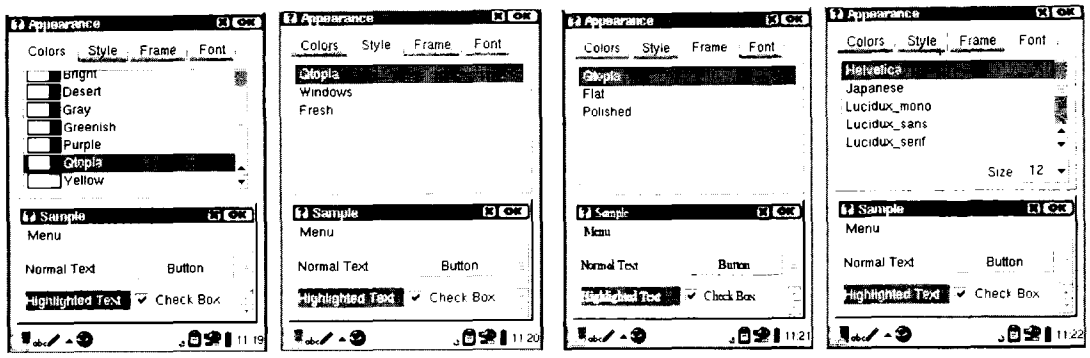


图 6-11 Qtopia 外观设置应用程序运行后的窗口 UI 界面

6.6.1 main 函数

Qtopia 外观设置应用程序的 main 函数通过宏定义完成，它调用主窗口部件类的构造函数 AppearanceSettings ，函数 main 的宏定义（在 src/setting/appearance/main.cpp 中）如下：

```
QTOPIA_ADD_APPLICATION("appearance",AppearanceSettings);
QTOPIA_MAIN
```

6.6.2 SampleWindow 类

SampleWindow 类从 QWidget 继承，它是外观设置显示的样例预览窗口。用户选择外观设置后，可以立即从 SampleWindow 窗口中看到新设置的外观效果。如果满意效果，则通过点击 OK 按钮更新所有窗口的外观；如果不满意，可重新选择外观设置。 SampleWindow 类窗口部件在 UI 界面中的位置如图 6-12 所示。

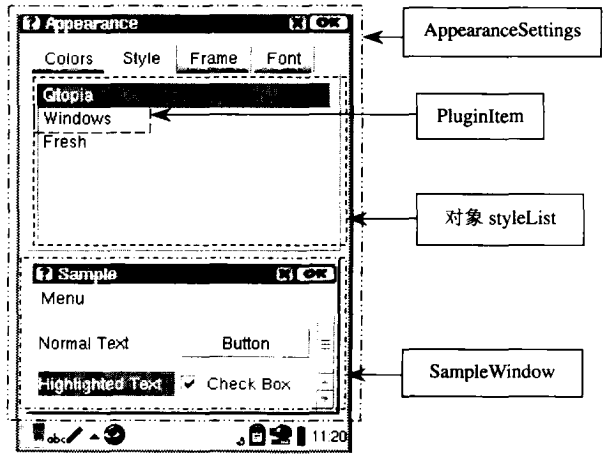


图 6-12 SampleWindow 类窗口部件在 UI 界面中的位置

SampleWindow 类的成员函数及其功能说明如表 6-21 所示。

表 6-21 SampleWindow 类的成员函数及其功能说明

成员函数名	功能说明
SampleWindow(QWidget *parent)	构造函数，调用函数 init()创建窗口部件
sizeHint()	返回窗口的尺寸

续表

成员函数名	功能说明
setFont(const QFont &f)	设置字体
setDecoration(WindowDecorationInterface *i)	设置接口为 i，计算出装饰区域
paintEvent(QPaintEvent *)	调用装饰插件接口函数，绘制装饰区域
setUpdatesEnabled(bool e)	设置所有窗口系统 UI 更新是否激活
resizeEvent(QResizeEvent *re)	重新计算窗口大小

6.6.3 AppearanceSettings 类

AppearanceSettings 类是外观设置应用程序的主窗口，用户通过它可以选择 UI 界面的各种外观设置，并可在 SampleWindow 窗口中看到设置的效果。该类的成员函数及其功能说明如表 6-22 所示。

表 6-22 AppearanceSettings 类的成员函数及其功能说明

成员函数名	功能说明
AppearanceSettings(QWidget* parent,...)	构造函数，构造外观设置应用程序的窗口
populateStyleList()	在 styleList 视图中按照风格插件名加入 ListItem（视图条目）
populateDecorationList()	在 decorationList 视图中按照装饰插件名加入 ListItem
populateColorList()	在 colorList 视图中，根据*.scheme 文件中的相关值加入 ListItem。根据 etc/themes/ .conf 文件的 Theme 组中的 name 值加入 ListItem
readColorScheme(int id)	从 etc/colors/*.scheme 文件中读出 colors 组中各项值，设置调色板颜色
populateFontList(const QString& cur, int cursz)	将字体映射链表 fontMap[i]中字体的名字及大小作为 ListItem 加入到 fontList 视图中
decorationSelected(int idx)	根据选择的装饰插件，设置 Sample 窗口部件中的 ListItem，并使用插件接口函数重画所有窗口
colorSelected(int id)[slot]	根据用户选择设置调色板颜色
styleSelected(int idx) [slot]	根据用户选择设置 Sample 窗口部件的新风格
setStyle(QWidget *w, QStyle *s)	将窗口 w 设置为新的风格 s
decorationSelected(int idx) [slot]	根据用户选择设置 Sample 窗口部件的新装饰
fontSelected(const QString &name) [slot]	根据用户选择设置 Sample 窗口部件的新字体
accept()[slot]	当用户按下 OK 按钮时，将用户选择的配置写入配置文件。并向系统信道发消息 applyStyle()，让每个应用程序更新风格且关闭窗口

6.6.4 AppearanceSettings 类的构造函数分析

AppearanceSettings 类用于 PDA 设备的 UI 界面，其构造函数分析如下：

```
AppearanceSettings::AppearanceSettings( QWidget* parent,  const char* name, WFlags fl )
    : AppearanceSettingsBase( parent, name, TRUE, fl|Qt::WStyle_ContextHelp )
{
    wdiface = 0;
    styleiface = 0;
    wdLoader = new PluginLoader( "decorations" );  //装载装饰插件
    styleLoader = new PluginLoader( "styles" );  //装载风格插件
    wdIsPlugin = FALSE;

   //字体最大尺寸限制
    maxFontSize = qApp->desktop()->width() >= 640 ? 24 : 12;
```

```
    sample = new SampleWindow( this );  //建立预览窗口部件
    AppearanceSettingsBaseLayout->addWidget( sample );
    populateStyleList();  //将风格插件的名字以 ListItem 加入到 StyleList 视图中
    populateDecorationList();//将装饰插件的名字以 ListItem 放入到 DecorationList 视图中
    populateColorList();//将*.scheme 文件对应颜色以 ListItem 加入到 ColorListStyleList 视图中

    //从配置文件中读出当前的颜色并设置对应 ListItem 为高亮选中状态
    Config config("qpe");
    config.setGroup( "Appearance" );
    QString s = config.readEntry( "Scheme", "Qtopia" );
    //根据用户选择，设置调色板颜色
    connect( colorList, SIGNAL(highlighted(int)),
                        this, SLOT(colorSelected(int)) );
    colorList->setCurrentItem( colorListIDs.findIndex(s) );

    //从配置文件中读出当前的风格并设置对应 ListItem 为高亮选中状态
    s = config.readEntry( "Style", "Qtopia" );
    unsigned i;
    if ( s == "QPE" ) s = "Qtopia";
    for ( i = 0; i < styleList->count(); i++ ) {
         PluginItem *item = (PluginItem*)styleList->item(i);
         if ( item->filename() == s || item->text() == s ) {
            styleList->setCurrentItem( i );
            break;
         }
    }
    connect( styleList, SIGNAL(highlighted(int)),
                     this, SLOT(styleSelected(int)) );

    //从配置文件中读出当前的装饰并设置对应 ListItem 为高亮选中状态
    s = config.readEntry( "Decoration" );
    for ( i = 0; i < decorationList->count(); i++ ) {
         PluginItem *item = (PluginItem*)decorationList->item(i);
         if ( item->filename() == s || item->text() == s ) {
            decorationList->setCurrentItem( i );
            break;
         }
    }
    connect( decorationList, SIGNAL(highlighted(int)),
                       this, SLOT(decorationSelected(int)) );
    //设置 Sample 窗口部件中装饰接口名，并重画 Sample 窗口
    decorationSelected( decorationList->currentItem() );

    //读出当前字体配置及字体大小
    s = config.readEntry( "FontFamily", "Helvetica" );
    prefFontSize = config.readNumEntry( "FontSize", font().pointSize() );
    populateFontList(s,prefFontSize);
    connect( fontSize, SIGNAL(activated(const QString&)),
                this, SLOT(fontSizeSelected(const QString&)) );
}
```

（1）更新 Sample 窗口的颜色

函数 colorSelected从配置文件中读取颜色值并更新绘制窗口部件，列出如下：

```
void AppearanceSettings::colorSelected( int id )
{
   QPalette pal = readColorScheme(id);
    //将 sample 中的各个窗口部件设置设色板为 pal，并更新绘制窗口部件
   sample->setPalette( pal );
}
```

在 $QTOPIA_PATH/etc/colors 目录下的颜色配置文件中保存有各种颜色的配置值，如 Blues.scheme，

其内容列出如下：

```
[Translation]
File=QtopiaColorSchemes
Context=ColorScheme
[Global]
Name[]=Blues
[Colors]
Background=#E4F1FF
Button=#C9DFFF
Highlight=#2861FF
AlternateBase=#DAE9FF
```

函数 readColorScheme 从参数 id 对应的颜色配置文件中读出颜色值或位图，为窗口部件的不同部分设置颜色或位图，列出如下：

```
QPalette AppearanceSettings::readColorScheme(int id)
{
   Config config( QPEApplication::qpeDir() + "etc/colors/" + colorListIDs[id] + ".scheme",
      Config::File );
   config.setGroup( "Colors" );
   //从颜色配置文件中读出颜色值或位图名，设置窗口各部分的颜色或位图
   QPalette tempPal;
   setPaletteEntry( tempPal, config, Button, "#F0F0F0" );
   setPaletteEntry( tempPal, config, Background, "#EEEEEE" );
   QPalette pal( tempPal.normal().button(), tempPal.normal().background() );
   setPaletteEntry( pal, config, Button, "#F0F0F0" );
   setPaletteEntry( pal, config, Background, "#EEEEEE" );
   setPaletteEntry( pal, config, Base, "#FFFFFF" );
   setPaletteEntry( pal, config, Highlight, "#8BAF31" );
   setPaletteEntry( pal, config, Foreground, "#000000" );
   //设置字体颜色
   QString color = config.readEntry( "HighlightedText", "#FFFFFF" );
   pal.setColor( QColorGroup::HighlightedText, QColor(color) );
   color = config.readEntry( "Text", "#000000" );
   pal.setColor( QColorGroup::Text, QColor(color) );
   color = config.readEntry( "ButtonText", "#000000" );
   pal.setColor( QPalette::Active, QColorGroup::ButtonText, QColor(color) );

   pal.setColor( QPalette::Disabled, QColorGroup::Text,
          pal.color(QPalette::Active, QColorGroup::Background).dark() );

   return pal;
}
```

函数 setPalEntry 从配置 config 中读出颜色值或位图 defaultVal ，并为窗口的组件 role 设置颜色，列出如下：

```
#define setPaletteEntry(pal, cfg, role, defaultVal) \
   setPalEntry(pal, cfg, #role, QColorGroup::role, defaultVal)
static void setPalEntry( QPalette &pal, const Config &config, const QString &entry,
                  QColorGroup::ColorRole role, const QString &defaultVal )
{
   QString value = config.readEntry( entry, defaultVal );
//如果颜色配置文件中配置项第一个字母含有#，表示是颜色值；否则表示为位图名
   if ( value[0] == '#' )     pal.setColor( role, QColor(value) );
   else
        pal.setBrush( role, QBrush(QColor(defaultVal), Resource::loadPixmap(value)) );
}
```

（2）更新 Sample 窗口的风格

风格是指窗口的样式，图 6-13 所示为 Qtopia 与 Windows 风格下 Sample 窗口的不同样式对照。从图中可以看出两种风格下的窗口标题栏、按钮及滚动条的样式不同。

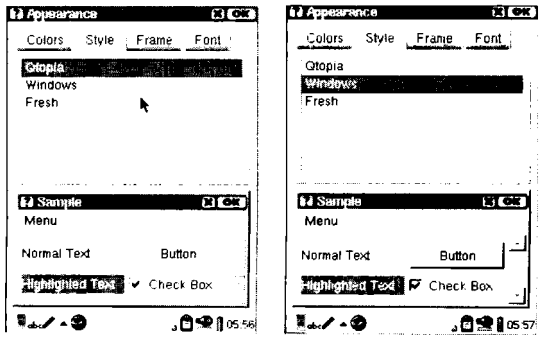


图 6-13　Qtopia 与 Windows 风格下 Sample 窗口的不同样式对照

函数 styleSelected 选择用户定义的风格类或风格插件并设置风格，列出如下：

```
void AppearanceSettings::styleSelected( int idx )
{
    QString style("Qtopia");
    PluginItem *item = (PluginItem *)styleList->item( idx );
    if ( item )
        style = item->filename().isEmpty() ? item->text() : item->filename();

    StyleInterface *oldIface = styleiface;
    QStyle *newStyle = 0;
    styleiface = 0;
    if ( style == "Windows" ) { // No tr
        newStyle = new QWindowsStyle;
    } else if ( style == "QPE" || style == "Qtopia" ) {
        newStyle = new QPEStyle;
    } else {  //查询插件库，得到风格类对象
        StyleInterface *iface = 0;
        if ( styleLoader->queryInterface( style, IID_Style,
            (QUnknownInterface**)&iface ) == QS_OK && iface ) {
            newStyle = iface->style();
            styleiface = iface;
        }
    }

    if (!newStyle)  newStyle = new QPEStyle;
    sample->setUpdatesEnabled( FALSE );
    QPixmapCache::clear();
setStyle( sample, newStyle );  //设置风格
//设置 Sample 窗口到固定尺寸，重画 Sample 窗口
    QTimer::singleShot( 0, this, SLOT(fixSampleGeometry()) );
    if ( oldIface ) styleLoader->releaseInterface( oldIface );

}
```

函数 QWidget::setStyle 设置窗口部件的 GUI 风格为参数 style。如果未设置 style，窗口部件将使用应用程序的风格，即用 QApplication::style()替代。设置一个窗口部件的风格对存在或将来的子窗口部件没有影响。这个函数用于预览窗口风格很有用，应用程序不应用使用它，而应该统一设置风格。

函数 QWidget::setStyle 分析如下：

```
void QWidget::setStyle( QStyle *style )
{
    QStyle& old  = QWidget::style();
```

```
    createExtra();
    extra->style = style;
    if ( !testWFlags(WType_Desktop) //  除了 Qtopia desktop 外
       && testWState(WState_Polished)) { // 已被风格对象润饰过
      old.unPolish( this );  // 在新的风格设置前，必须取消旧的风格设置
      QWidget::style().polish( this );  //调用新的用户风格函数设置风格
    }
    styleChange( old ); //当风格改变时，更新绘制窗口部件
}
```

不同的风格类有不同的润饰函数 polish，函数 polish 根据窗口部件 w 的类型及继承关系，在 w 的父类中设置窗口的线条、调色板及阴影等窗口元素。 Flat 风格类的函数 polish 列出如下：

```
void FlatStyle::polish( QWidget *w )
{
    //设置 QFrame 类的阴影及线条宽度
    if ( w->inherits( "QFrame" ) ) { //如果 w 是从 QFrame 继承
         QFrame *f = (QFrame *)w;
         if ( f->frameShape() == QFrame::HLine || f->frameShape() == QFrame::VLine )
            f->setFrameShadow( QFrame::Plain );
         else if ( f->frameShape() != QFrame::NoFrame )
            f->setFrameShape( QFrame::StyledPanel );
         f->setLineWidth( 1 );
    }

    if ( w->inherits( "QSpinBox" ) )
          ((SpinBoxHack*)w)->setFlatButtons( TRUE );

    if ( w->inherits( "QMenuBar" ) ) {
         // 定制菜单的 flat 外观
         QPalette p = w->palette();
         QColorGroup a = p.active();
         a.setColor( QColorGroup::Light, a.foreground() );
         a.setColor( QColorGroup::Dark, a.foreground() );
         p.setActive( a );
         p.setInactive( a );
         w->setPalette( p );
         w->installEventFilter( d );
    } else if ( w->inherits( "QHeader" ) ) {
         // 定制 headers 的 flat 外观
         QPalette p = w->palette();
         QColorGroup a = p.active();
         a.setColor( QColorGroup::Light, a.button() );
         p.setActive( a );
         p.setInactive( a );
         w->setPalette( p );
         w->installEventFilter( d );
    }
    QStyleHackWidget::setGlobalBrushOrigin(w);  //设置画刷起点
}
```

FlatStyle 类还重载了 QStyle 的各种绘制函数，如 drawItem、drawPanel、drawToolButton、drawComboButton 及 drawSlider 等。它们是窗口部件的基本绘制函数，可以直接改变窗口部件的图形及颜色等。在窗口更新或重画时会调用函数 paintEvent，Sample 窗口重载函数 paintEvent 调用风格插件的基本绘制函数。函数 paintEvent 分析如下：

```
void SampleWindow::paintEvent( QPaintEvent * )
{
    if ( !iface )    return;

    QPainter p( this );
    p.translate( lb, th+tb );  //座标系统 x+lb, y+th+tb
```

```
    iface->drawArea(WindowDecorationInterface::Border, &p, &wd);//绘制边框
    iface->drawArea(WindowDecorationInterface::Title, &p, &wd); //绘制标题

    //设置字体及颜色
    p.setPen(palette().active().color(QColorGroup::HighlightedText));
    QFont f( font() );
    f.setWeight( QFont::Bold );
    p.setFont(f);
    iface->drawArea(WindowDecorationInterface::TitleText, &p, &wd);
    //绘制 Help、OK、Close 按钮
    QRect brect( 0, -th, iface->metric(WindowDecorationInterface::HelpWidth,&wd), th );
    iface->drawButton( WindowDecorationInterface::Help, &p, &wd,
          brect.x(), brect.y(), brect.width(), brect.height(), (QWSButton::State)0 );
    brect.moveBy( wd.rect.width() -
           iface->metric(WindowDecorationInterface::OKWidth,&wd) -
           iface->metric(WindowDecorationInterface::CloseWidth,&wd), 0 );
    iface->drawButton( WindowDecorationInterface::Close, &p, &wd,
           brect.x(), brect.y(), brect.width(), brect.height(), (QWSButton::State)0 );
    brect.moveBy( iface->metric(WindowDecorationInterface::CloseWidth,&wd), 0 );
    iface->drawButton( WindowDecorationInterface::OK, &p, &wd,
           brect.x(), brect.y(), brect.width(), brect.height(), (QWSButton::State)0 );
}
```

（3）更新 Sample 窗口的装饰

Sample 窗口的装饰更新与风格的更新原理一样，只是装饰插件着重于为窗口加入了一些位图，如 OK 按钮使用位图代替。

例如，装饰风格类 FlatDecoration 重载了 QStlye 类的 drawArea 及 drawButton 函数，装饰风格类 FlatDecoration 使用前后的 Sample 窗口对比如图 6-14 所示。

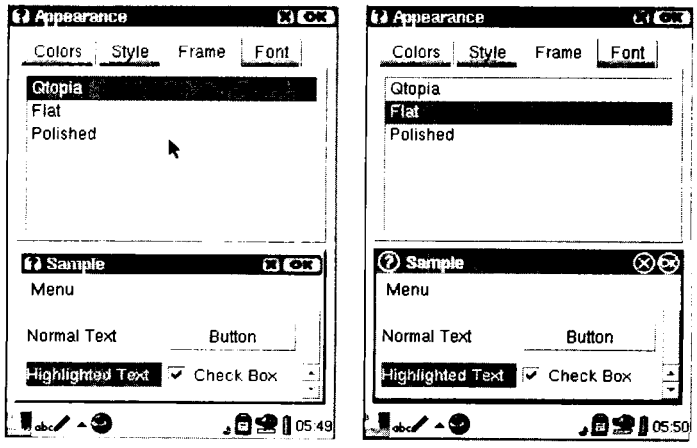


图 6-14　FlatDecoration 装饰类使用前后的 Sample 窗口对比

函数 decorationSelected 选择装饰风格类或装饰风格插件并设置装饰风格，列出如下：

```
void AppearanceSettings::decorationSelected( int idx )
{

    if ( wdIsPlugin ) {
        wdLoader->releaseInterface( wdiface );
    } else {
        delete wdiface;
    }
    wdiface = 0;
    wdIsPlugin = FALSE;
```

```
    QString dec("Qtopia");
    PluginItem *item = (PluginItem *)decorationList->item( idx );
    if ( item )
        dec = item->filename().isEmpty() ? item->text() : item->filename();

    if ( dec != "Qtopia" ) {
      if ( wdLoader->queryInterface( dec, IID_WindowDecoration,
            (QUnknownInterface**)&wdiface ) == QS_OK && wdiface )
        wdIsPlugin = TRUE;
    }

    if ( !wdiface )
           wdiface = new DefaultWindowDecoration;
    sample->setDecoration( wdiface );
    sample->repaint(); //重画 sample 窗口
}
```

函数 setDecoration 计算并设置装饰区域尺寸，以便绘制装饰区域，列出如下：

```
void SampleWindow::setDecoration( WindowDecorationInterface *i )
{
    iface = i; //将装饰插件给 iface，这样函数 paintEvent 可以调用其绘制函数
    wd.rect = QRect( 0, 0, 150, 75 );
    wd.caption = tr("Sample");
    wd.palette = palette();
    wd.flags = WindowDecorationInterface::WindowData::Dialog |
           WindowDecorationInterface::WindowData::Active;
    wd.reserved = 1;

    th = iface->metric(WindowDecorationInterface::TitleHeight, &wd);
    tb = iface->metric(WindowDecorationInterface::TopBorder, &wd);
    lb = iface->metric(WindowDecorationInterface::LeftBorder, &wd);
    rb = iface->metric(WindowDecorationInterface::RightBorder, &wd);
    bb = iface->metric(WindowDecorationInterface::BottomBorder, &wd);

    int yoff = th + tb;
    int xoff = lb;

    wd.rect.setX( 0 );
    wd.rect.setWidth( width() - lb - rb );
    wd.rect.setY( 0 );
    wd.rect.setHeight( height() - yoff - bb );

    container->setGeometry( xoff, yoff, wd.rect.width(), wd.rect.height() );
    setMinimumSize( QMIN(container->sizeHint().width()+lb+rb, qApp->desktop()->width()-10),
             QMAX(SAMPLE_HEIGHT,container->sizeHint().height()+tb+th+bb) );
}
```

（4）更新外观设置

设置 AppearanceSettings 窗口风格并点击 OK 按钮后，风格被通过发消息通知各个应用程序更新风格的方式应用到所有的窗口，图 6-15 所示为默认的 Qtopia 风格与用户设置风格的窗口对比，用户设置的风格颜色为 Yellow，风格设置为 Windows，Frame （即装饰）设置为 Flat。

函数 accept 是窗口按响应 OK 按钮时的槽函数，它完成窗口的关闭及窗口参数的保存工作。AppearanceSettings 类重载了 accept 函数，函数 accept 将 AppearanceSettings 窗口中设置的风格及颜色参数保存到配置文件中。对 Qtopia 窗口系统应用这个风格，调用父类的 accept 函数继续完成窗口。

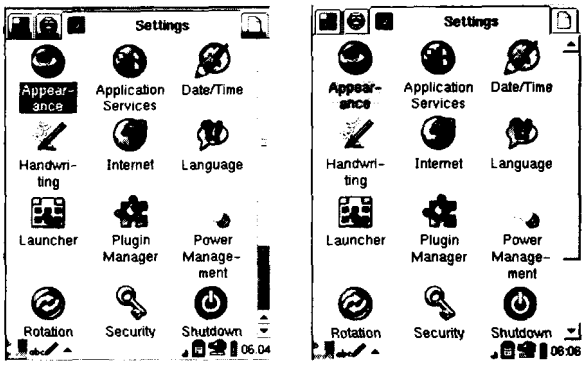


图 6-15　默认的 Qtopia 风格与用户设置风格的窗口对比

函数 accept 分析如下：

```
void AppearanceSettings::accept()
{
    Config config("qpe");
    config.setGroup( "Appearance" );
    QString s;
   //从视图条目 PluginItem 得到风格名，并保存到 qpe.conf 中
    PluginItem *item = (PluginItem *)styleList->item( styleList->currentItem() );
    if ( item ) {
         s = item->filename().isEmpty() ? item->text() : item->filename();
         config.writeEntry( "Style", s );
    }

    //从视图条目得到装饰风格名，并保存到 qpe.conf 文件中
    item = (PluginItem *)decorationList->item( decorationList->currentItem() );
    if ( item ) {
         s = item->filename().isEmpty() ? item->text() : item->filename();
         config.writeEntry( "Decoration", s );
    }
    /*从视图条目中得到颜色名序号，从数组中查到颜色名，并将颜色参数存入到类似 blue.scheme 的颜色配置文件中*/
    s = colorListIDs[colorList->currentItem()];
    config.writeEntry( "Scheme", s );

    Config scheme( QPEApplication::qpeDir() + "etc/colors/" + s + ".scheme",
                   Config::File );
    if (scheme.isValid()){
         scheme.setGroup("Colors");
         QString color = scheme.readEntry( "Background", "#EEEEEE" );
         ……
         color = scheme.readEntry( "Shadow", "" );
         config.writeEntry("Shadow", color);
    } else {
         ……
    }
    //将字体配置写到 qpe.conf 文件中
    QFont font(fontList->currentText(), fontSize->currentText().toInt());
    config.writeEntry( "FontFamily", fontList->currentText() );
    config.writeEntry( "FontSize", fontSize->currentText().toInt() );

    QFontMetrics fm(font);
    config.writeEntry( "IconSize", QPE_FONT_HEIGHT_TO_ICONSIZE(fm.height()) );

    config.write(); //将 config 对象数据刷新到文件 qpe.conf 中
    //发消息将风格应用到 Qtopia 的所有窗口中
    Global::applyStyle();
```

```
    QDialog::accept(); // 调用父类 accept 函数继续完成窗口关闭工作
}
```

6.7 安装与卸载应用程序

安装与卸载应用程序通过应用程序 qipkg 来实现，它可以从本地或网上安装应用程序，还可以卸载应用程序。qipkg 应用程序的 UI 界面以向导形式指导用户一步步地完成安装或卸载操作，同时显示安装或卸载的进度、软件包的详细信息及帮助信息等。

qipkg 应用程序的设计思想是利用 Qt Designer 工具生成 UI 界面类 PackageWizardBase，然后在其子类 PackageWizard 中实现 UI 界面操作功能。qipkg 应用程序最终使用命令 ipkg 来安装或卸载应用程序。

PackageWizardBase 类是用 Qt Designer 工具绘制的 UI 界面，它只是直观地绘制了界面图形并加上了 connect 函数连接信号与槽，用户可在 Qt Designer 工具中预览生成的 UI 界面。Qt Designer 工具绘制的 UI 界面用 XML 语言描述，保存在以 .ui 结尾的文件中。类 PackageWizardBase 存在 qkwizard.ui 中。

qipkg 应用程序运行时的 UI 界面图如图 6-16 所示。

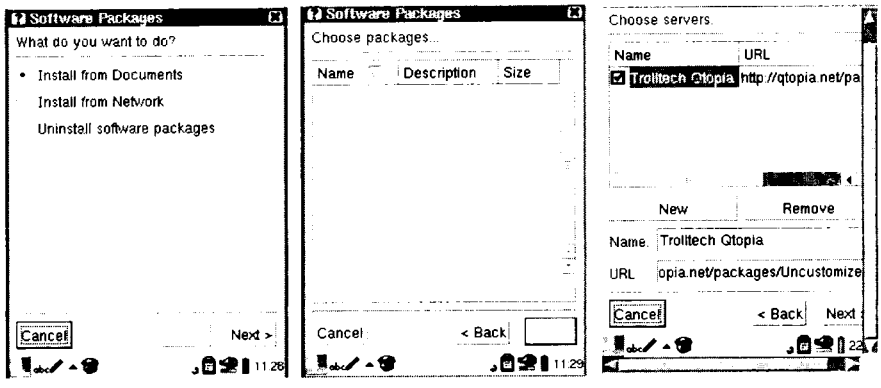


图 6-16 qipkg 应用程序运行时的 UI 界面图

qipkg 应用程序的 UI 界面操作代码较简单，对照应用程序的运行即将各个函数与每步完成的操作对应起来，这样就容易理解 PackageWizard 类。成员函数 readSettings()在从网上安装应用程序时，被用来从 etc/ipkg.conf 中读出网址、名字及安装路径，并以 ListItem 形式加到 Server 列表视图中，如图 6-16 中的第 3 幅图。函数 writeSettings()将 Server 列表视图中的 ListItem 条目的网址、名字及安装路径信息保存到配置文件 etc/ipkg.conf 中。

构造函数 PackageWizard 用来设置由 Qt designer 工具绘制的 UI 界面类 PackageWizardBase 中各个控件的初始状态，该函数列出如下：

```
PackageWizard::PackageWizard( QWidget* parent,  const char* name, WFlags fl )
    : PackageWizardBase( parent, name, fl )
{
    infoProcess = 0;
    committed = FALSE;
    installedRootDict = 0;

    connect( newserver, SIGNAL(clicked()), this, SLOT(newServer()) );
    connect( removeserver, SIGNAL(clicked()), this, SLOT(removeServer()) );
    connect( servers, SIGNAL(currentChanged(QListViewItem*)), this, SLOT(editServer(QListViewItem*)) );
    servername->setEnabled(FALSE);
```

```
    serverurl->setEnabled(FALSE);

    if (!pm_uninstalled) {//设置图标
        pm_uninstalled = new QPixmap(Resource::loadPixmap("uninstalled"));
        pm_installed = new QPixmap(Resource::loadPixmap("installed"));
        pm_install = new QPixmap(Resource::loadPixmap("install"));
        pm_uninstall = new QPixmap(Resource::loadPixmap("uninstall"));
    }
//设置 packagelist 列表视图属性
    packagelist->setColumnWidthMode(0,QListView::Manual);
    packagelist->setColumnWidthMode(1,QListView::Manual);
    packagelist->setColumnWidthMode(2,QListView::Manual);
    packagelist->setColumnAlignment(2,AlignRight);
    packagelist->setSelectionMode( QListView::Multi );
    packagelist->setVScrollBarMode(QScrollView::AlwaysOn);
    packagelist->installEventFilter(this);
    packagelist->setShowSortIndicator(TRUE);

    ipkg_old = 0;
    readSettings();  //读取网络配置，设置 server 列表视图
    updateServerSelection();
    progress_net->hide();
    progress_confirm->hide();
    QPEApplication::setStylusOperation(packagelist->viewport(),QPEApplication::RightOnHold);
    //package_description = new DetailsPopup(this);
    delete mode_share; // ### Not implemented yet
    //设置各个窗口中 Help 按钮的激活状态
    setHelpEnabled(page_mode,FALSE);
    setHelpEnabled(page_servers,FALSE);
    setHelpEnabled(page_packages,FALSE);
    setHelpEnabled(page_share,FALSE);
    setHelpEnabled(page_location,FALSE);
    setHelpEnabled(page_confirmation,FALSE);
}
```

6.8 AppServices 应用程序

AppServices 应用程序用来显示 Qtopia 提供的各种服务及这些服务对应的各种应用程序，同时显示打开各种 MIME 类型的应用程序名，用户可以选择打开 MIME 类型的文档的应用程序。AppServices 应用程序类似于 Windows 操作系统选择文件打开类型的“打开”对话框。

Qtopia 提供了一些特定的服务应用程序，如 Alarm（闹钟）及 Contact（通信录）等服务。与一般的应用程序不同，它依赖于具体的设备配置。为了提供设备配置的灵活性，设备的一项服务可能有许多应用程序来支持，如可能有多个应用程序支持 Alarm 服务，因此 Qtopia 在 AppServices 应用程序中允许用户选择支持某项服务的应用程序。

Qtopia 提供的各种服务的应用程序的配置说明保存在$QTOPIA_PATH/sevices 目录下，如 Alarm 服务应用程序的配置文件为 Alarm.service，它类似于一般应用程序的 desktop 配置文件。同时提供了服务对应的应用程序，以一个目录表示，如 Alarm 目录。这个目录下存在提供这项服务的应用程序名对应的文件名，如有一个 clock 文件，clock 文件名表示对应的应用程序名也为 clock。而应用程序在 bin 目录下，Alarm 目录下的 clock 只是应用程序 clock 的配置说明，如版本号等。

同样，某一 MIME 类型的文档可能被多个应用程序打开，Qtopia 在 AppServices 应用程序中允许用户选择打开某一 MIME 类型文档的应用程序。在每个应用程序的配置文件 desktop 中有其支持的 MIME 类型，Qtopia 通过 AppLnk 类及 AppLnkSet 类获得 desktop 文件中的配置信息。AppServices 应用程序根据 AppLnkSet 对象中的对象信息，保存各种 MIME 类型。并且支持 MIME 类型的应用程序名用 ASCheckListItem 对象来表示，这样用户即可选择打开 MIME 类型的文档的应用程序。

当用户在 AppServices 应用程序完成选择并点击 Ok 按钮后，如果存在多个打开应用程序，会把选择的默认应用程序名存在类似于 Sevice-alarm（对于 Alarm 服务来说）的配置文件中 sevice 组的 default 项上。

AppServices 应用程序的主窗口类是 AppServices 类，该类从 AppServicesBase 类继承。AppServicesBase 类是由 Qt designer 工具绘制的 UI 界面，其各窗口部件的控制在 AppServices 类中完成。AppServices 应用程序运行时显示的窗口及窗口部件类如图 6-17 所示。

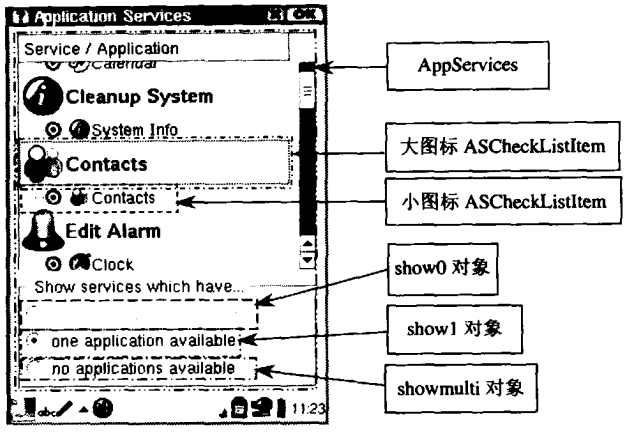


图 6-17　AppServices 应用程序运行时显示的窗口及窗口部件类

实现 AppServices 应用程序的两个类是 AppServices 和 ASCheckListItem 类，本节分别对这两个类进行说明。

6.8.1　AppServices 类

AppServices 类完成选择打开文件类型应用程序的各种操作，服务或文件类型（MIME 类型）及打开文件的应用程序名在 AppServices 中以 ASCheckListItem 对象表示。AppServices 类的构造函数调用函数 loadState() 来装载 ASCheckListItem 对象。

函数 loadState()根据配置文件设置服务名或 MIME 类型名及应用程序名，以 ASCheckListItem 对象装载到从 QListBox 继承的视图中，分析如下：

```
void AppServices::loadState()
{
    lv->clear();  //QListView 视图清空
    drawWait(TRUE);//装载时显示等待图标
    delete allapps;
    /*从$QTOPIA_PATH/apps 下读出所有的.desktop 文件中配置，每个.desktop 对应一个 AppLnk 对象，所有 AppLnk 对象组成
AppLnkSet 链表*/
    allapps=new AppLnkSet(MimeType::appsFolderName());

    int max=-1;
    QDict<ASCheckListItem> servicedict;  //对象链表

//绘制小图标的 QCheckListItem 对象代表用于打开的应用程序
#define ADDAPP(si,lnk,ver) \
    ASCheckListItem* ai = new ASCheckListItem(si,lnk); \
    /*ai->setText(1,ver);*/ \
    ai->setPixmap(0,lnk->pixmap()); \
    if ( def == lnk->exec() ) { \
        ai->setOn(TRUE); \
    }
```

```
        /*处理所有的服务*/
        //list()通过递归得到$QTOPIA_PATH/sevices 下的所有目录名（去除路径及后缀）
        QStringList s = Service::list();
        for (QStringList::ConstIterator it = s.begin(); it!=s.end(); ++it) {
           //sv为目录名+路径，*it为目录名,例如：$QTPIA_PATH/sevices/Alarm
             QString sv=QPEApplication::qpeDir()+"services/"+(*it);
             QDir dir(sv,QString::null,QDir::Name | QDir::IgnoreCase,QDir::Files);
             QStringList apps = dir.entryList();//得到一个目录下的文件列表
           /*找到目录名对应的.service配置文件,读配置文件中的 name 项。例如：Edit Alarm 以其为图标名创建对象,如创建 Edit Alarm
的对象*/
             ASCheckListItem* si = new ASCheckListItem(lv,(*it),0);
             if ( si->isValid() ) {
                QString def;
              /*binding() 函数仅在*it 前加上 Sevice-，如*it为 alarm 时，函数返回 Sevice-alarm，构造配置文件 Sevice-alarm
的Config 对象.*/
                Config binding(Service::binding(*it));

               /*对于存在多个服务应用程序的情况，读取默认的服务应用程序名*/
               /*如果Config 对象binding 有效,即找到配置文件时,读取 Sevice 组 default值,大多数情况下没有类似于 Sevice-alarm
的配置文件*/
                if ( binding.isValid() ) {
                     binding.setGroup("Service")   ;
                     def = binding.readEntry("default");
                }
                servicedict.insert(*it,si); //将 ListItem 对象加入到链表中保存

               //遍历目录中的文件，查找这些文件名对应的应用程序的.desktop配置文件
                for (QStringList::ConstIterator ait = apps.begin(); ait!=apps.end();
    ++ait) {
                  /*在 AppLnkSet 中查找名为*ait的 AppLnk对象,如*ait为 clock,其对应的应用程序的配置文件为 clock.desktop,
AppLnk对象从这个配置文件产生。成员函数 name()返回值 Clock*/
                     const AppLnk* lnk = allapps->findExec(*ait);
                     if ( lnk ) {
                      //例如：$QTOPIA_PATH/sevices/Alarm/clock
                        QFile f(sv+"/"+*ait);
                        if ( f.open(IO_ReadOnly) ) {
                        /*将对应应用程序的.desktop 中的name项值设为图标名，如 clock.desktop 文件的 name 项值为 Clock，将
其设为QCheckListItem对象   上标签名*/
                         //创建带 RadioButton 的 QCheckListItem对象
                        ADDAPP(si,lnk,ver);
                     }
                     }
                }
                si->setOpen(TRUE);
             } else {
                delete si;
             }
        }

        /*处理所有 MIME类型*/
        // 将所有的 MIME 类型以带 Open标签的大图标 QCheckListItem 对象加入视图中
        //得到各个应用程序对应的 AppLnk 即对应应用程序的.desktop 文件
        const QList<AppLnk> &c = allapps->children();
        const AppLnk* lnk;
       //遍历每个应用程序对应的 AppLnk
        for (QListIterator<AppLnk> ai(c); (lnk=ai.current()); ++ai) {
             QStringList types = lnk->mimeTypes()  ;
            //遍历每个 AppLnk 对应的.desktop 文件中的所有 mimetype
             for (QStringList::ConstIterator ti=types.begin(); ti!=types.end(); ++ti)
            {
              //如：mt 为application/ipkg
                QString mt = *ti;
                if ( mt.right(2)=="/*" )
                  mt.truncate(mt.length()-2); //  去掉末尾的“/*”
                QString sv = "Open/"+mt;  // Open/变成类似于 application/ipkg
```

```
        ASCheckListItem* s = servicedict.find(sv);
        if ( !s ) {// 如果未找到，即对应的 QCheckListItem对象不存在
          //创建带 Open 的 QCheckListItem  对象，如 Open ipkg
            s = new ASCheckListItem(lv,sv,lnk);
            servicedict.insert(sv,s); //放入对象链表中
            s->setOpen(TRUE);
        }

       //对于存在多个打开应用程序的情况，读取默认打开的应用程序名
        Config binding(Service::binding(sv));
        QString def;
        if ( binding.isValid() ) {
            binding.setGroup("Service");
            def = binding.readEntry("default")   ;
        }
       /*将 AppLnk 对应的.desktop 文件中 name项作为图标标签名用小图标绘制 QCheckListItem 对象*/
        ADDAPP(s,lnk,"1.00");
    }
}
/* 设置 show0、show1、showmulti控件状态 */
bool some = FALSE;
QDictIterator<ASCheckListItem> sdit(servicedict);
ASCheckListItem* si;
//遍历 QCheckListItem 对象链表
for (; (si=sdit.current()); ++sdit) {
   int n = si->childCount();
    max = QMAX(max,n);
    if ( n==0 && show0->isOn()
       || n==1 && show1->isOn()
       || n>1 && showmulti->isOn())
    {
       some = TRUE;
       if ( n == 1 ) {//只有一个用于打开的应用程序时，第 1 个为选中状态
            ((ASCheckListItem*)si->firstChild())->setOn(TRUE);
       }
    } else {
       delete si;
    }
}

if ( !some ) {
    switch ( max ) {
       case -1:
            //没有服务
            choices->setEnabled(FALSE);
            break;
       case 1:
            //没有选择服务
            showmulti->setEnabled(FALSE);
            if ( showmulti->isChecked() ) {
               show1->setChecked(TRUE);
               loadState();  //递归装载
            }
            break;
       case 0:
            //没有提供服务
            showmulti->setEnabled(FALSE);
            show1->setEnabled(FALSE);
            if (!show0->isChecked()) {
               show0->setChecked(TRUE);
               loadState();//递归装载
            }
            break;
    }
```

```
    }
    drawWait(FALSE);//停止画 等待图标
}
```

当用户在窗口中选择并点击 Ok 按钮 后，会执行 done 函数。如果打开的应用程序可以是多个时，函数 done 将用户选择的应用程保存在 sevice-*配置文件中，如 sevice-alarm 配置文件。在下次装载 AppServices 窗口时，可以根据 sevice-*配置文件来设置窗口中的应用程序的默认选中状态。函数 done 列出如下：

```
void AppServices::done(int y)
{
    if ( y ) {
        if ( changed ) {
            QListViewItem* s = lv->firstChild();
            while ( s ) {
                ASCheckListItem* sv=(ASCheckListItem*)s;
              //遍历视图中所有的 ASCheckListItem 对象
                for (QListViewItem* a = s->firstChild(); a; a = a->nextSibling())
{
                    ASCheckListItem* ap=(ASCheckListItem*)a;
                  //如果打开应用程序多于 1个，写配置文件 servie-*
                    if ( ap->isOn() && sv->childCount()>1 ) {
                        Config binding(Service::binding(sv->service())) ;
                        binding.setGroup("Service")  ;
                        binding.writeEntry("default",ap->application())   ;
                        break;
                    }
                }
                s = s->nextSibling();// 下一个 ASCheckListItem 对象
            }
        }
        // Could list, but server ignores at the moment
        QCopEnvelope e("QPE/System","serviceChanged(QString)");
        e << QString::null;
    }
    AppServicesBase::done(y);
    close();
}
```

6.8.2 ASCheckListItem 类

ASCheckListItem类从 QCheckListItem 类继承， QCheckListItem 类有两个构造函数，其参数不同。构造函数 ASCheckListItem(QListView *, const QString, const AppLnk*)创建大图标，用于显示服务或 MIME 类型的 QCheckListItem 对象；另一个构造函数 ASCheckListItem(QCheckListItem, const AppLnk)创建小图标，用于显示打开的应用程序名的 QCheckListItem 对象。这两个构造函数分析如下：

```
ASCheckListItem(QListView *parent, const QString &sv, const AppLnk* lnk) :
    QCheckListItem(parent,QString::null,Controller), link(lnk), svr(sv)
{
   //例如 sv为 contact，则 config(sv)返回$QPOPIA_PATH/sevices/contact.service
    Config service(Service::config(sv),Config::File);
    QString icon;
  //对于 Qtopia 系统提供的服务，将服务名称用 QCheckListItem对象大图标表示
    if ( service.isValid() ) {
      //从.service 配置文件中读出 Service 组 Name项的值
        service.setGroup("Service") ;
        QString nm = service.readEntry("Name");
        if ( nm.isEmpty() || service.readNumEntry("Multiple",0) ) {
            //不显示字符给用户看
            svr = QString::null;
```

```
        } else {
            setText(0,nm); //ListItem 图标名称设置为配置文件中的 Name 项值
            icon = service.readEntry("Icon");//从配置文件中读出 Icon 项值
        }
    }
    //用 QCheckListItem 对象大图标显示 MIME 类型后缀，如.txt
    else if ( lnk ) {//如果 lnk 参数存在
        //去掉一些前缀，例如：sv 为 open/text
        QString name=sv.mid(5); // 去掉" Open/"
        if ( name.left(12)=="application/" ) {
            QString ap = name.mid(12);   //  去掉"application/"
            if ( ap.left(2) == "x-" )
                ap = ap.mid(2); // 去掉"x-"
            name = ap;
        } else if ( name.find('/')>0 ) {
            //如果后缀为 MimeType 扩展名，如.txt，以扩展名为图标名，如.TXT
            MimeType mt(name);
            QString ext = mt.extension().upper() ;
            if ( !ext.isEmpty() )
                name = ext;
        } //设置图标名，例如：open video
        setText(0,AppServices::tr("Open %1").arg(name));
        icon = lnk->icon();//得到图标
    } else {
        svr = QString::null;
    }
    int s = AppLnk::bigIconSize(); //使用大图标
    QImage img;
    if ( !icon.isNull() ) // 装载图像
        img = Resource::loadImage(icon);
    if ( img.isNull() ) // 图像为空，装载默认图像
        img = Resource::loadImage("qpe/Generic");
    img = img.smoothScale(s,s);
    QPixmap pm; pm.convertFromImage(img);//转换成位图
    setPixmap(0,pm); // 设置图标
}

ASCheckListItem(QCheckListItem *parent, const AppLnk* lnk) :
    QCheckListItem(parent,lnk->name(),RadioButton), link(lnk)
{
}
```

6.9 插件管理器

Qtopia 插件安装后分类放在 $QTOPIA_PATH/plugins 目录下，如属于 applet 类的插件放在 $QTOPIA_PATH/plugins/applets 目录下。每一类插件在其目录下有一个 .dirctory 文件描述该类插件的属性，例如，applet 类插件的 dirtory 文件（在 $QTOPIA_PATH/plugins/applets/.dirctory 中）内容列出如下：

```
[Translation]
File=QtopiaPlugins
Context=Applets
[Desktop Entry]
Name[]=Applets
Comment[]=Applets provide status information and easy access to settings via the Taskbar.
Apply=QPE/TaskBar reloadApplets()
```

每一个插件都以 .so 文件格式保存并且有一个 .desktop 配置文件对其进行说明，如 clock 插件的配置文件 clockapplet.desktop 的内容列出如下：

```
[Desktop Entry]
```

```
Name=Clock
Comment=Displays a clock in the Taskbar.
#日本文字的名字
Name[ja]=æ™,è¨ˆ
#日本文字的描述
Comment[ja]=ã,¿ã,¹ã,¯ãƒŠãƒ¼ä¸…ã□§æ™,è¨ˆã,'è¡¨ç¤ºã□—ã□¾ã□™ã€,
X-QPE-Library=libclockapplet.so
```

插件管理器 pluginmanager 应用程序用视图窗口显示所有插件，用户可以选中插件。选中的插件将被装载，未选中的插件将被卸载，插件可以是窗口部件类，也可以是不带窗口部件的编/解码引擎。插件的装载/卸载由插件装载器 PluginLoader 类完成，pluginmanager 应用程序通过 PluginManager、PluginTypeItem 和 PluginItem 共 3 个类实现窗口操作，其中 PluginTypeItem 和 PluginItem 类从 QCheckListItem 类继承。pluginmanager 应用程序运行时的窗口及窗口部件类如图 6-18 所示。

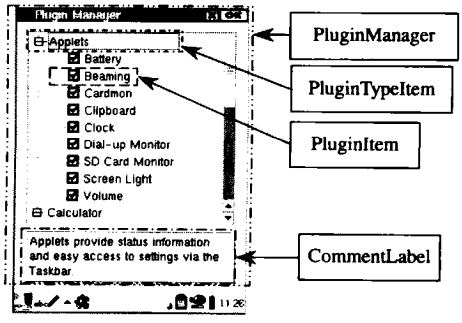


图 6-18 pluginmanager 应用程序运行时的窗口及窗口部件类

PluginManager 类从 PluginManagerBase 类继承，PluginManagerBase 类是用 Qt designer 绘制的窗口，窗口部件的功能在子类 PluginManagerBase 中实现。PluginManager 类构造函数调用 init 函数显示插件视图，创建 QVBoxLayout 及 CommentLabel 对象显示插件的描述。函数 init 从.dirctory 配置文件中读出每类插件的配置，并创建每类插件的 PluginTypeItem 对象显示在视图中。函数 init 从每个插件的 .desktop 配置文件中读出配置，创建每个插件的 PluginItem 对象显示在视图中。并从插件装载器 PluginLoader 对象中读出插件的激活情况，设置 PluginItem 对象的选中状态。

当用户从 PluginManager 类窗口上选择或取消插件并点击 Ok 按钮后，PluginManager 类的 save() 函数被调用。函数 save()会利用插件装载器 PluginLoader 激活或取消插件并发送消息给插件，使其从 UI 界面中退出或加到 UI 界面上。

第 7 章　Qtopia 的 sysinfo 应用程序分析

sysinfo 应用程序用图形窗口的形式显示各种系统信息，如软件版本号、文件系统存储信息、 CPU 的使用信息、内存信息及各种类型文档大小信息等。这些信息分别使用 QTabWidget 类子窗口显示，sysinfo 应用程序的主窗口是 SystemInfo 类，该类创建 QTabWidget 对象。 sysinfo 应用程序运行显示的窗口及窗口类如图 7-1 所示。

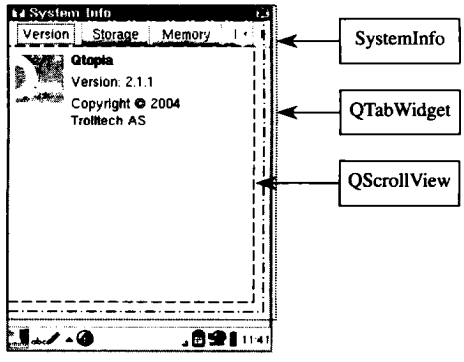


图7-1　sysinfo 应用程序运行显示的窗口及窗口类

本章详细分析 sysinfo应用程序的源代码。

sysinfo应用程序通过/proc文件系统得到各种系统信息，通过 FileSystem 类通过 C 库函数 statfs得到硬盘的使用信息。

sysinfo应用程序的 main 主函数用 Qtopia 的统一的宏来声明，这样应用程序既可以作为单独的应用程序启动，也可以作为共享库被快速启动器启动。 sysinfo 应用程序的 main 主函数的宏定义（在 sysinfo/main.cpp 中）列出如下：

```
QTOPIA_ADD_APPLICATION("sysinfo",SystemInfo)
QTOPIA_MAIN
```

在 sysinfo 应用程序中，软件版本信息用窗口部件类 VersionInfo 显示，文件系统存储信息用窗口部件类 StorageInfoView 显示，CPU 的运行信息用窗口部件类 LoadInfo 显示，内存的使用信息用窗口部件类 MemoryInfo 显示，各种文档占用空间信息用窗口部件类 DataView 显示，主窗口部件类 SystemInfo 将这些信息显示窗口组织在一起。

7.1　SystemInfo 类

SystemInfo 类通过构造函数创建 sysinfo 应用程序的各窗口部件及 QTabWidget 对象，并使用 QTabWidget 对象管理每种信息的窗口部件。SystemInfo 类的构造函数列出如下：

```
SystemInfo::SystemInfo( QWidget *parent, const char *name, WFlags f )
    : QWidget( parent, name, f ), wizard(0)
```

```
{
    QScrollView *sv;
    //设置窗口图标
    setIcon( Resource::loadPixmap( "SystemInfo" ) );
    setCaption( tr("System Info") );//设置标题
    QVBoxLayout *lay = new QVBoxLayout( this ); //创建布局
    tab = new QTabWidget( this );
    lay->addWidget( tab ); //将窗口部件tab加入到布局中

    //创建软件版本信息窗口部件
    sv = new QScrollView(this, "VersionSV");
    sv->setHScrollBarMode(QScrollView::AlwaysOff);
    sv->setResizePolicy(QScrollView::AutoOneFit);
    sv->setFrameStyle(QFrame::NoFrame);
    sv->addChild( new VersionInfo( tab ) );
    tab->addTab( sv, tr("Version") );

#if defined(_OS_LINUX_) || defined(Q_OS_LINUX)
    //创建文件系统存储信息窗口部件
    sv = new QScrollView(this, "StorageSV");
    QWidget *w = new QWidget(sv->viewport()); //在视图的viewport()视口中创建窗口部件
    sv->setHScrollBarMode(QScrollView::AlwaysOff); //设置为不需要滚动条
    sv->setResizePolicy(QScrollView::AutoOneFit);
    sv->setFrameStyle(QFrame::NoFrame); //设置框架模式
    sv->addChild(w); //将窗口部件w加入到视图中

    QVBoxLayout * vl = new QVBoxLayout(w); //创建窗口部件w的布局
    vl->addWidget( new StorageInfoView(w) );
    //按下按钮时进入清除存储空间向导操作
    QPushButton * cleanup = new QPushButton(tr("Cleanup..."), w, "start cleanup");
    cleanup->setSizePolicy(QSizePolicy(QSizePolicy::Maximum, QSizePolicy::Maximum));
    connect(cleanup, SIGNAL(clicked()), this, SLOT(startCleanupWizard()));
    vl->addWidget( cleanup ) ;
    QSpacerItem* spacer = new QSpacerItem( 20, 20, QSizePolicy::Minimum,
          QSizePolicy::Expanding);
    vl->addItem(spacer);

    connect(qApp, SIGNAL(appMessage(const QCString&, const QByteArray&)),
          this, SLOT(appMessage(const QCString&, const QByteArray&)));

    tab->addTab(sv, tr("Storage"));
#endif

#ifdef SYSINFO_GEEK_MODE
    tab->addTab( new LoadInfo( tab ), tr("CPU") );
    //tab->addTab( new Graphics( tab ), tr("Graphics") );
#endif

    //创建内存的使用信息窗口部件
    tab->addTab( new MemoryInfo( tab ), tr("Memory") );
    //创建各种文档占用空间信息窗口部件
    tab->addTab( new DataView( tab ), tr("Data") );
    resize( 220, 180 );
}
```

7.2 VersionInfo 类

VersionInfo 类从 QWidget 类继承，用于显示 Linux 内核版本号及 Qtopia 版本号，该类的窗口及窗口部件类对象如图 7-2 所示。

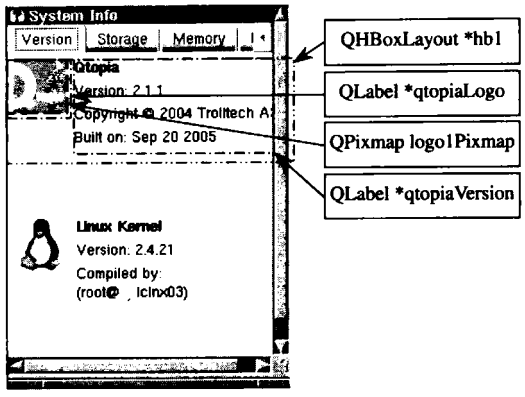


图 7-2 VersionInfo 类的窗口及窗口部件类对象

VersionInfo 类的构造函数创建了版本信息显示窗口的各个部件类，列出如下：

```
VersionInfo::VersionInfo( QWidget *parent, const char *name, WFlags f )
    : QWidget( parent, name, f )
{
    if (QApplication::desktop()->width() > 240)
        setMinimumSize( 200, 150 );

    QVBoxLayout *vb = new QVBoxLayout( this, 4 );
    QString lineBreak;
#ifdef QTOPIA_PHONE
    lineBreak = "<br>";
#else
    Q_UNUSED(lineBreak);
#endif

    //得到 Linux 内核版本号及编译者
#ifdef SYSINFO_GEEK_MODE
    QString kernelVersionString;
    QFile file( "/proc/version" );
    if ( file.open( IO_ReadOnly ) ) {
        QTextStream t( &file );
        QString v;
        //每次>>操作读出一个字符串，这里只需要第 3个字符串
        t >> v; t >> v; t >> v;
        v = v.left( 20 );  //例如字符串为“2.4.21”
        kernelVersionString = tr( "<b>Linux Kernel</b><p>Version:%1 " ).arg(lineBreak) + v + "<p>";
        t >> v;
        kernelVersionString += tr( "Compiled by: " ) + v;
        file.close();
    }
#endif

    //得到 Qtopia版本号及编译日期
    QString qtopiaVersionString;
    qtopiaVersionString = tr( "<b>Qtopia</b><p>Version: " ) + Global::version() + "<p>";
    qtopiaVersionString += tr( "Copyright &copy; %1 %2%3", "%1 = 'year'  %2 = 'conditional linebreak - ignore
for translation', %3 = 'company'" )
                    .arg(2004).arg(lineBreak)
                    .arg("Trolltech AS");
#ifdef SYSINFO_GEEK_MODE
    qtopiaVersionString += "<p>" + tr( "Built on: %1" ).arg(lineBreak) + __DATE__ ;
#endif

    //将 Qtopia版本号字符串显示在窗口上
    QHBoxLayout *hb1 = new QHBoxLayout( vb );
```

```
    hbl->setSpacing( 2 );

    double imageScale = ((double)QApplication::desktop()->width())/290.0;
    if (imageScale > 1.0) //得到图像缩放比例
        imageScale = 1.0;
    int imageSize = (int)(55 * imageScale);
    int finalSize = (int)(60 * imageScale);
    QLabel *qtopiaLogo = new QLabel( this );
    QImage logo1 = Resource::loadImage( "qpe-logo" );
    logo1 = logo1.smoothScale( imageSize , imageSize );
    QPixmap logo1Pixmap;
    logo1Pixmap.convertFromImage( logo1 ); //转换成位图
    qtopiaLogo->setPixmap( logo1Pixmap );
    qtopiaLogo->setFixedSize( finalSize, finalSize );//设置固定大小
    hbl->addWidget( qtopiaLogo, 0, Qt::AlignTop + Qt::AlignLeft );

    QLabel *qtopiaVersion = new QLabel( this );
    qtopiaVersion->setText( qtopiaVersionString );
    hbl->addWidget( qtopiaVersion, 1, Qt::AlignTop + Qt::AlignLeft );
    hbl->setSpacing( 2 );

    //将 Linux 内核版本号字符串显示在窗口上
#ifdef SYSINFO_GEEK_MODE
    QHBoxLayout *hb2 = new QHBoxLayout( vb );
    QLabel *linuxLogo = new QLabel( this );

    /*使用 image 不需要写额外的 qpainter 代码，但由于直接转换到位图时，所以它不进行 alpha 混合，因此使用了 qpainter 代码*/
    QPixmap logo2Pixmap( finalSize , finalSize );
    QColor bgColor = colorGroup().background();
    QPainter painter( &logo2Pixmap );
    painter.fillRect( QRect( 0, 0, finalSize, finalSize ), QBrush( bgColor ) );
    QImage logo2 = Resource::loadImage( "tux-logo" );
    logo2 = logo2.smoothScale( imageSize, imageSize );
    painter.drawImage( 0, 0, logo2 );
    painter.end();
    linuxLogo->setPixmap( logo2Pixmap );
    linuxLogo->setFixedSize( finalSize, finalSize );
    hb2->addWidget( linuxLogo, 0, Qt::AlignTop + Qt::AlignLeft );

    QLabel *kernelVersion = new QLabel( this );
    kernelVersion->setText( kernelVersionString );
    hb2->addWidget( kernelVersion, 1, Qt::AlignTop + Qt::AlignLeft );
#endif
}
```

7.3 StorageInfoView 类

StorageInfoView 类从 QWidget 类继承，它用图形化形式显示各个文件系统的存储容量信息，并且可以通过清除向导来清除存储空间。StorageInfoView 类窗口及其子窗口类分布图如图 7-3 所示。

StorageInfoView 类调用了函数 updateMounts()，该函数更新所有挂接的文件系统的信息及窗口上的显示。函数 updateMounts() 分析如下：

```
void StorageInfoView::updateMounts()
{
    const QList<FileSystem>& sifs(sinfo->fileSystems());//得到对象链表
    QListIterator<FileSystem> sit(sifs);

    int n = sifs.count();
    delete vb;
```

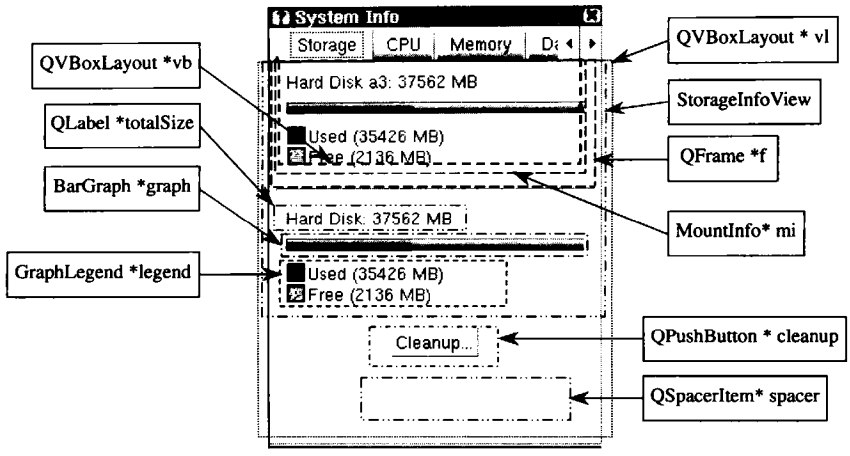


图 7-3 StorageInfoView 类窗口及其子窗口类分布

```
    vb = new QVBox( this );
    vb->setSpacing( n > 3 ? 1 : 5 );
    bool frst=TRUE;
    FileSystem* fs;
    //遍历所有文件系统
    for ( ; (fs=sit.current()); ++sit ) {
        if ( !frst ) {
            QFrame *f = new QFrame( vb );
            f->setFrameStyle( QFrame::HLine | QFrame::Sunken );
            f->show() ;
        } else frst=FALSE;
        //每个文件系统使用一个 MountInfo 部件显示
        MountInfo* mi = new MountInfo(fs, vb);
        connect(this, SIGNAL(updated()), mi, SLOT(refresh()));
    }

    if ( n < 3 ) // add a filler
        (new QWidget(vb))->setSizePolicy( QSizePolicy(QSizePolicy::Expanding,
          QSizePolicy::Expanding))  ;
    setVBGeom();//设置 QVBox 对象尺寸
    updateGeometry();//更新窗口尺寸
    vb->show();// 显示 QVBox 对象 vb
}
```

（1）MountInfo 类

MountInfo 类从 QWidget 继承，其构造函数创建 QLabel 对象显示文件系统逻辑盘的总容量，创建 BarGraph 对象用一个条形条显示逻辑盘的已用和空闲空间。并且创建 GraphLegend 对象，分别用图例方式显示已用和空闲空间的大小。 MountInfo 类的构造函数列出如下：

```
MountInfo::MountInfo( const FileSystem* f, QWidget *parent, const char *name )
    : QWidget( parent, name ), title(f->name())
{
    fs = f;
    //创建 QVBoxLayout 垂直排列窗口部件，各个窗口部件之间空隙为 7 个像数点
    QVBoxLayout *vb = new QVBoxLayout( this, 7 )    ;

    //创建 QLabel 对象显示文件系统逻辑盘总容量字符串
    totalSize = new QLabel( this );
    vb->addWidget( totalSize );

    data = new GraphData();
```

```
    graph = new BarGraph( this );
    graph->setFrameStyle( QFrame::Panel | QFrame::Sunken );
    /*放大因子为 1，表示比例为 1。窗口部件在没有伸展因素下被布置到一个布局中时，窗口部件一般根据 QWidget::sizePolicy()分
配尺寸。放大因子用来根据窗口部件互相的比例改变其被分配的尺寸*/
    vb->addWidget( graph, 1 );
    graph->setData( data );//设置数据

    legend = new GraphLegend( this );
    vb->addWidget( legend );
    legend->setData( data );

    refresh();
}
```

SystemInfo 类的成员函数 updateMounts()将信号 updated()与 MountInfo 类槽函数 refresh()连接起来，函数 refresh()计算出文件系统的存储容量并转换为一定格式的字符串显示在窗口系统中。该函数列出如下：

```
void MountInfo::refresh()
{
    long mult = 0;
    long div = 0;
    if ( fs->blockSize() ) {  //硬盘的块尺寸
         mult = fs->blockSize() / 1024;
         div = 1024 / fs->blockSize();
    }
    if ( !mult ) mult = 1;
    if ( !div ) div = 1;
    long total = fs->totalBlocks() * mult / div;
    long avail = fs->availBlocks() * mult / div;
    long used = total - avail;
    data->clear();
    if (total < 10240 ) {//以KB为单位显示
       totalSize->setText( title + ": " + tr("%1 kB").arg( total ) );
       data->addItem( tr("Used (%1 kB)").arg(used), used );
       data->addItem( tr("Free (%1 kB)").arg(avail), avail );
    }
    else {//以MB为单位显示
       totalSize->setText( title + ": " + tr("%1 MB").arg( total/1024 ) );
       data->addItem( tr("Used (%1 MB)").arg(used/1024), used );
       data->addItem( tr("Free (%1 MB)").arg(avail/1024), avail );
    }
    graph->repaint( FALSE );
    legend->update();
    graph->show();
    legend->show();
}
```

GraphData 类用字符串链表 names 保存各个图画的名字，使用整数链表 values 来保存每个图画代表的数值，并提供函数 clear()清除链表。函数 addItem（const QString &name, int value） 在链表中加入成员；函数 name(int i)得到图画的名字；函数 value(int i)得到图画代表的数值；函数 count()得到链表中成员的个数，即图画的个数。

BarGraph 类和 GraphLegend 类都使用 QPainter类来画图显示逻辑盘的容量数据，二者所用方法类以，这里只分析 GraphLegend 类。

（2）GraphLegend 类

GraphLegend 类从 QFrame 类继承，它用图例方式显示逻辑盘的已用或空闲空间，该类的完整代码列出如下：

```
GraphLegend::GraphLegend( QWidget *parent, const char *name, WFlags f )
```

```
    : QFrame( parent, name, f )
{
    horz = FALSE;
}
//设置字体排列方向
void GraphLegend::setOrientation(Orientation o)
{
    horz = o == Horizontal;
}
//画上图画及字符串
void GraphLegend::drawContents( QPainter *p )
{
    int total = 0;
    //i 为 0 时，value 值为逻辑盘已使用的空间；i=1 时，value 为空闲空间值
    for ( unsigned i = 0; i < data->count(); i++ )
                total += data->value(i);//得到逻辑盘的总容量值

    int tw = width()/data->count()-1;
    int th = height()/(horz ? 1 : data->count());
    if ( th > p->fontMetrics().height() )
              th = p->fontMetrics().height(); //设为字体的高度
    int x = 0;
    int y = 0;
    for ( unsigned i = 0; i < data->count(); i++ ) {
         QColor col;
         //设置颜色的 hue（色调）、saturation（饱和度）、value(亮度值)
         col.setHsv( i * 360 / data->count(), 255, 255 );
         p->setBrush( col );//设置画刷颜色
         p->drawRect( x+1, y+1, th - 2, th - 2 );//画矩形区域
         //画上文字
         p->drawText( x+th + 1, y + p->fontMetrics().ascent()+1, data->name(i) );
         if ( horz ) {
            x += tw;
         } else {
            y += th;
         }
    }
}
//计算窗口部件的大小
QSize GraphLegend::sizeHint() const
{
    int th = fontMetrics().height() + 2;  //文字的高度
    int maxw = 0;
    for ( unsigned i = 0; i < data->count(); i++ ) {
         int w = fontMetrics().width( data->name(i) );//字符串的宽度
         if ( w > maxw )
            maxw = w;
       }
       if ( 0 && horz ) {
            return QSize( maxw * data->count(), th );
       } else {
         return QSize( maxw, th * data->count() );
    }
}
//设置每个图画的名字及代表的数值
void GraphLegend::setData( const GraphData *p )
{
    data = p;
    int th = fontMetrics().height();
    setMinimumHeight( th * ( horz ? 1 : data->count() ) );
    updateGeometry();
}
```

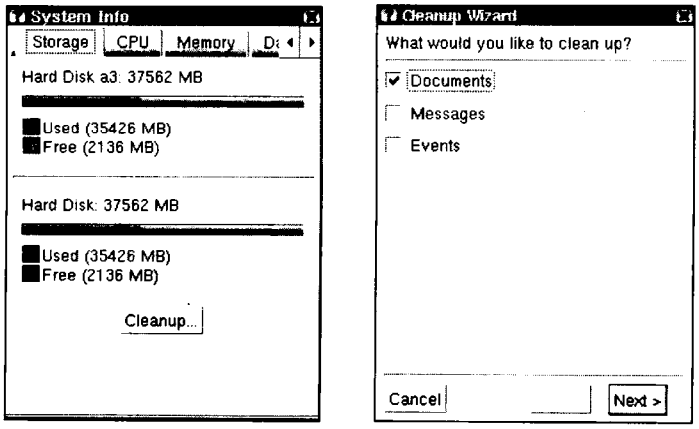
（3）CleanupWizard 类

CleanupWizard 类从 QWizard 类继承，当用户在 SystemInfo 窗口中按下 Cleanup 按钮后，该类将提供清除向导指导用户一步步地清除存储空间。 CleanupWizard 类的构造函数创建 PreselectionWidget 类对象、DocCleanWidget类对象、DocResultWidget 类对象、DocSummaryWidget 类对象、MailCleanWidget 类对象、DatebookCleanWidget 类对象和 FinalCleanupWidget 类对象。

Cleanup 按钮如图 7-4（a）所示。

➢ PreselectionWidget 类

PreselectionWidget 类从 QWidget 继承，其构造函数执行后的窗口显示如图 7-4（b）所示。



（a）

图 7-4 Cleanup 按钮及 PreselectionWidget 类的构造函数执行后的显示

➢ DocCleanWidget 类

DocCleanWidget 类从 QWidget 类继承，其构造函数执行后显示的窗口如图 7-5 所示。DocCleanWidget 类的实现比较简单，这里不加分析。

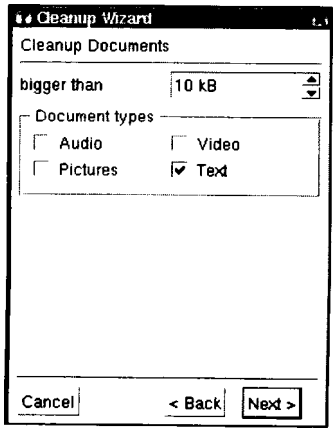


图 7-5 DocCleanWidget 类的构造函数执行后显示的窗口

➢ DocResultWidget 类

DocResultWidget 类从 QWidget 类继承，使用 QListView 来显示并清除搜索到的文档。

构造函数 DocResultWidget(...) 创建 DocResultWidget 类的各窗口部件对象，构造函数执行后显示的窗口如图 7-6 所示。

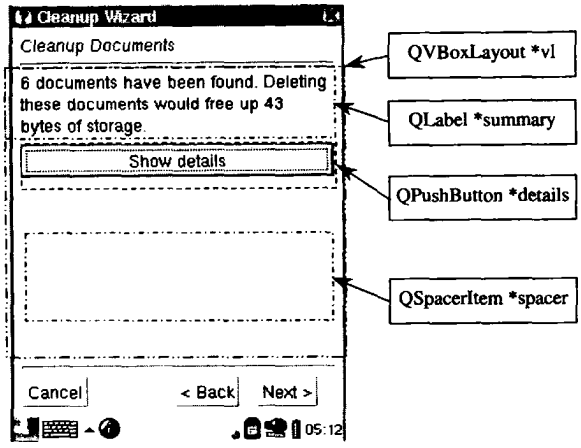


图 7-6　DocResultWidget 类构造函数执行后显示的窗口

DocResultWidget 类的几个主要函数分析如下。

函数 performSearch()查找符合过滤类型的文档，并将满足大小要求的文档用 DocCheckListItem 对象显示在 QListView 视图中。同时在标签 summary 显示查找到与文档个数相关的字符串。函数 performSearch()列出如下：

```
void DocResultWidget::performSearch()
{
    list->clear();// 清除 QListView 视图
    DocLnk *dl;
    DocLnkSet allDocs;
    selectedFileSize = 0;
    uint hits = 0;
    QFontMetrics fmt = list->fontMetrics ();
    //filter是在 DocCleanWidget 类窗口中选择的文件类型，即 audio/* 或 video/* 或 text/*或 image/*
    //遍历选择的文件类型
    for (QStringList::ConstIterator fit=filter.begin(); fit != filter.end(); ++fit)
    {
      DocLnkSet resultDocs;
      /*从系统文档目录或可移去文件系统逻辑盘上过滤匹配 mimefilter 的文档,并添加文档的 DocLnk 对象到 resultDocs 上.*/
      Global::findDocuments( &resultDocs, *fit );

      const QList<DocLnk> doclist = resultDocs.children ();
      QListIterator<DocLnk> it(doclist);
      DocCheckListItem *item = 0;
      QString unit;
      int displayedSize;
      //遍历过滤得到的各个文档 DocLnk 对象，将满足大小要求的文档显示在视图中
      while ((dl=it.current())) {
          QFileInfo fInfo(dl->file());
          //minFileSize是 DocCleanWidget类窗口中设置的最小尺寸
          if (minFileSize*1024 < (int)fInfo.size()){
               item = new DocCheckListItem(list,
                         dl->name(), QCheckListItem::CheckBox, *dl);
              //计算选中文件的总容量
              selectedFileSize += fInfo.size(); //忽略链接文件的大小（很小的尺寸）
              item->setOn(TRUE);
              sizeUnit(fInfo.size(), unit, &displayedSize);//设置文件大小单位
              bool isMB = FALSE;
              if ( fInfo.size()/1024 > 10000 )
                        isMB = TRUE;
              item->setText(1, tr("%1 %2").arg(displayedSize).arg(unit));
              list->insertItem(item);  //插入到 QListView 视图
```

```
            connect(item, SIGNAL(selectionChanged(int)),
                          this, SLOT(selectionChanged(int)) );
            hits++;//选择的文档计数
          }
          ++it;
       }
    }
    //在 QLabel 对象 summary 上显示找到的文档的个数相关的字符串
    if (!hits) {
          summary->setText(tr("<qt>There are no documents on this device"
               " that match the given filter critieria.</qt>"));
          list->hide();
          details->hide();
          summary->setFocus();//update ContextBar
     } else {//没有找到文档
        QString unit;
        int displayedSize = 0;
        sizeUnit(selectedFileSize, unit, &displayedSize);

        summary->setText(tr("<qt>%1 documents have been found. "
                    "Deleting these documents would free up %2 %3 of "
                    "storage.</qt>", "%1 = number, %2 = number, %3 = MB/kB/Bytes" )
                   .arg(hits)
                   .arg(displayedSize).arg(unit));

        details->setText(tr("Show details"));
        details->disconnect();
        connect(details, SIGNAL(clicked()), this, SLOT(showDetails()));
        list->hide();
        details->setFocus();
        details->show();
    }
};
```

函数 cleanup()删除 QListView 视图中选中的文档并清空视图，列出如下：

```
void DocResultWidget::cleanup()
{

    DocCheckListItem *item = (DocCheckListItem *) list->firstChild();
   if (!item) //对象不存在、空的
              return;

   qDebug("Deleting Documents   ");
   while (item && !cleanupStopped) {
      if (item->isOn()) {
         QFileInfo info(item->docLink().file());
         emit docDeleted(item->docLink().name(), info.size());
         qApp->processEvents(); //进行事件处理
         item->docLink().removeFiles();  //删除文件
         item->setOn(FALSE); // uncheck to keep track what was done
       }
      item = (DocCheckListItem *) item->nextSibling() ;
    }

    if (cleanupStopped)  //删除停止
            emit docDeleted(tr("Aborted"), -1);
    else {// 删除完成
       emit docDeleted(tr("Done"), -1);
       list->clear();  // 清空 QListView 视图
    }
};
```

函数 selectionChanged 是槽函数，当在 QListView 视图中的 DocCheckListItem 对象的选中状态发

生变化时,会发出信号触发函数 selectionChanged 运行。该函数计算文件大小的单位,并在标签 summary 上给出提示字符串，列出如下：

```
void DocResultWidget::selectionChanged(int size)
{
   selectedFileSize += size;
   QString unit;
   int displayedSize;
   sizeUnit(selectedFileSize, unit, &displayedSize);
   summary->setText(tr("<qt>Uncheck documents you want to "
                       "keep (%1 %2 selected)</qt>")
                    .arg(displayedSize).arg(unit));
};
```

➢ DocCheckListItem 类

DocCheckListItem 类从 QCheckListItem 类继承，DocCheckListItem 用来在 QListView 视图中显示文档名、图标及文档大小。DocCheckListItem 类在窗口中的位置如图 7-7 所示。

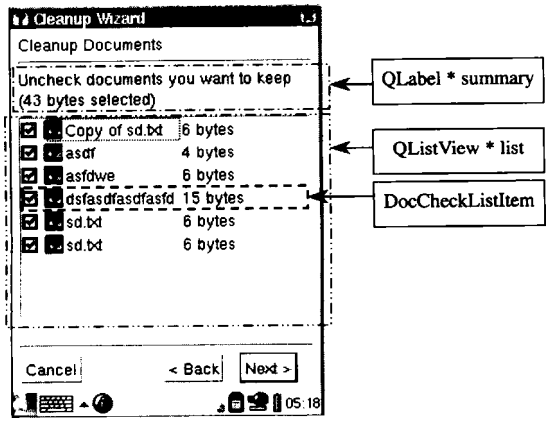


图 7-7　DocCheckListItem 对象在窗口中的位置

当文档的 DocCheckListItem 对象的选中状态发生变化时，函数 stateChange 被触发来发出信号 selectionChanged，该函数列出如下：

```
void stateChange(bool b) {
         int size = QFileInfo(dlk.file()).size();//得到文档的大小
         if (b)//被选中
            emit selectionChanged(size);//增加文档的大小 size
         else
            emit selectionChanged(-size); //减小文档的大小 size
      };
```

➢ DocSummaryWidget 类

DocSummaryWidget 类使用进度条显示删除文档的进度，并显示删除过程中的提示信息。在删除的过程中，可以用 Cancel 按钮来取消删除过程。删除后显示删除信息的标签控件隐藏起来， Cancel 按钮变成 Done 按钮。DocSummaryWidget 类在运行中的窗口显示如图 7-8 所示。

DocSummaryWidget 类主要使用函数 monitorFileDeletion和函数 docDeleted 来改变删除文档时的窗口状态，下面对这两个函数分别进行说明。

函数 monitorFileDeletion 设置进度条的总进度数、按钮及进度条的状态，列出如下：

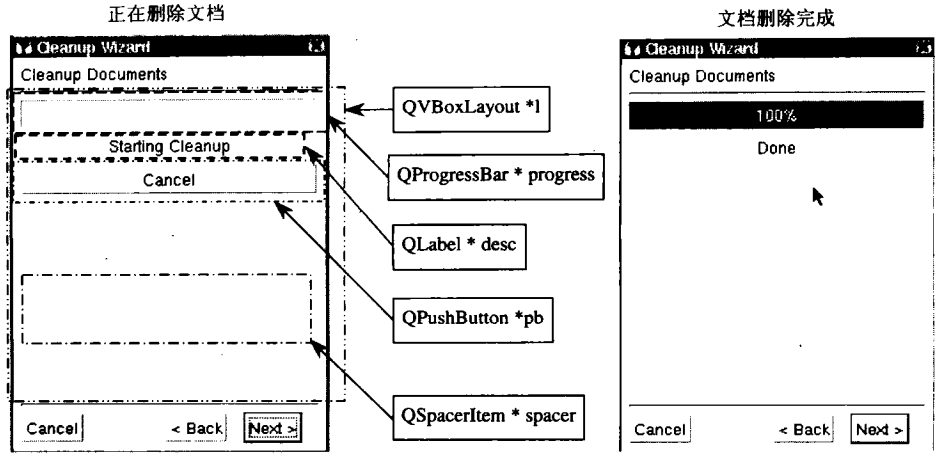


图 7-8 DocSummaryWidget 类在运行中的窗口显示

```
void DocSummaryWidget::monitorFileDeletion(int fileSize)
{
    if (fileSize) {
       pb->setText(tr("Cancel"));
       pb->show();// 显示 Cancel 按钮
       desc->setText(tr("Starting Cleanup"))  ;
       progress->reset();  //重置进度条
       //设置进度条的总进度数为所有删除文件的大小和
       progress->setTotalSteps(fileSize);
       progress->show();
    } else {
       pb->hide();  //隐藏 Cancel 按钮
       progress->hide();// 隐藏进度条
       desc->setText(tr("<qt>No action was selected.</qt>"));
       desc->setFocus(); //设置键盘焦点
    }
};
```

函数 docDeleted 在每个文件被删除时设置进度条的进度、标签及按钮上的状态。每个文件被删除时，该函数被调用一次，列出如下：

```
void DocSummaryWidget::docDeleted(const QString& fileName, int fileSize)
{
    static int docCount = 0;
    if (!deleted)
       docCount = 0;
    if (fileSize > 0)
    {
       docCount++;
       deleted+=fileSize;
       //设置进度条的进度，按文件的大小增加进度
       progress->setProgress( deleted );
       desc->setText(tr("Deleting... ") + fileName);
    } else {
       desc->setText(fileName);
       pb->hide();
       desc->setFocus(); //update ContextBar
       if (deleted)  //删除完成
             emit docCleanupFinished(docCount);
    }
};
```

➢ MailCleanWidget 类

MailCleanWidget 类从 QWidget 类继承，用来删除过期的邮件，该类的构造函数执行后显示的窗口及其组成部件类如图 7-9 所示。

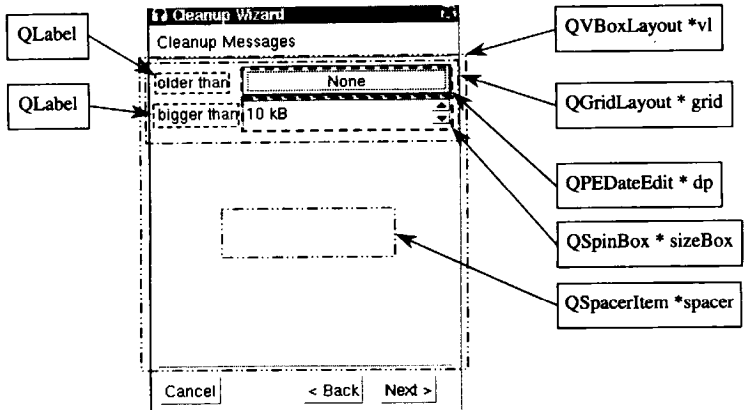


图7-9 MailCleanWidget 类的构造函数执行后显示的窗口及其组成部件类

➢ FinalCleanupWidget 类

FinalCleanupWidget 类从类 QWidget 继承，它用标签显示删除的结果。该类的构造函数运行后显示的窗口如图 7-10 所示。

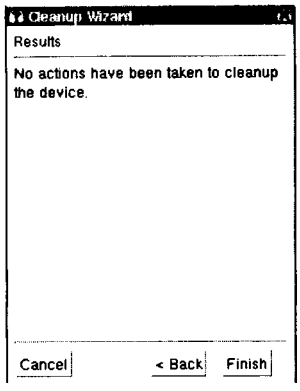


图7-10 FinalCleanupWidget 类的构造函数执行后显示的窗口

7.4 负载信息 LoadInfo

LoadInfo 类从 QWidget 类继承，其构造函数创建 LoadInfo 窗口中的各个窗口部件，其中 Load 类对象定时绘制 CPU 负载图。该类的窗口显示如图 7-11 所示。

LoadInfo 类成员函数 getCpuInfo 通过读取文件/proc/cpuinfo 得到 CPU 的信息，以显示在 LoadInfo 窗口的标签中。函数 getCpuInfo 列出如下：

```
QString LoadInfo::getCpuInfo()
{
   QString info = tr("Type: ");
#ifdef SYSINFO_GEEK_MODE
   bool haveInfo = FALSE;
   QFile f( "/proc/cpuinfo" );
```

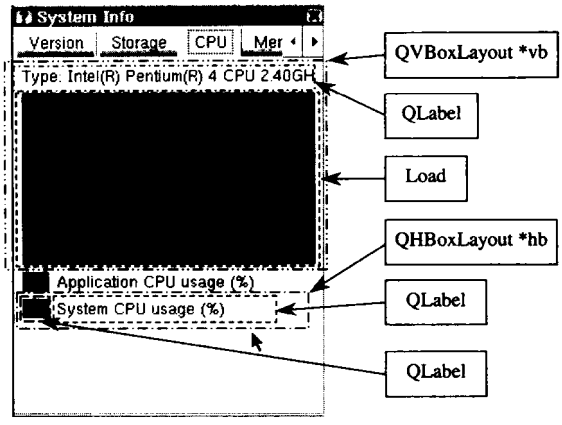


图 7-11　LoadInfo 类的窗口显示

```
    if ( f.open( IO_ReadOnly ) ) {
        QTextStream ts( &f );  //字符流

        while ( !ts.atEnd() ) {  //读出 CPU 的模型、频率及处理器名
            QString s = ts.readLine();
            if ( s.find( "model name" ) == 0 ) { // No tr
                info += s.mid( s.find( ':' ) + 2 );
                haveInfo = TRUE;
            } else if ( s.find( "cpu MHz" ) == 0 ) {
                double mhz = s.mid( s.find( ':' ) + 2 ).toDouble();
                info += " " + QString::number( mhz, 'f', 0 );
                info += "MHz";
                break;
            } else if ( s.find( "Processor" ) == 0 ) { // No tr
                info += s.mid( s.find( ':' ) + 2 );
                haveInfo = TRUE;
                break;
            }
        }
    }

    if ( !haveInfo )
        info = QString();
#endif
    return info;
}
```

Load 类构造函数初始化负载值，为定时读取 CPU 负载创建定时器，列出如下：

```
Load::Load( QWidget *parent, const char *name, WFlags f )
    : QWidget( parent, name, f ), lastUser(0), lastSys(0)
{
#ifdef SYSINFO_GEEK_MODE
    setMinimumHeight( 30 );//设置最小高度为 30
    setBackgroundColor( black );//设置背景为黑色
    points = 100;
    setMinimumWidth( points );  //设置最小宽度为 100
    userLoad = new double [points];
    systemLoad = new double [points];
    for ( int i = 0; i < points; i++ ) {
        userLoad[i] = 0.0;
        systemLoad[i] = 0.0;
    }
    maxLoad = 1.3; //最大负载为 130%
```

```
    //设置定时器，每隔 2000 ms 查看一次 CPU 负载
    QTimer *timer = new QTimer( this );
    connect( timer, SIGNAL(timeout()), SLOT(timeout()) );
    timer->start( 2000 );
    gettimeofday( &last, 0 ); // 调用 C 库函数得到 0时区的时间值存在 last 中
    first = TRUE;
    timeout();
#endif
}
```

函数 paintEvent 根据计算的负载值绘制负载曲线图，列出如下：

```
void Load::paintEvent( QPaintEvent * )
{
#ifdef SYSINFO_GEEK_MODE
    QPainter p( this );

    int h = height() - 5;
    int mult = (int)(h / maxLoad);  // 得到在图画中的负载平均高度

    p.setPen( gray );  // 设置为 gray (灰色) 的画笔
    p.drawLine( 0, h - mult, width(), h - mult ); //   绘制负载为 100%处的刻度线
    p.drawText( 0, h - mult, "100" ); // 绘制刻度标识文字 '100'
    p.drawText( 0, h, "0" );  //绘制刻度标识文字 '0'

    p.setPen( green );  //设置绿色的画笔，在整个画面上绘制系统负载曲线
    for ( int i = 1; i < points; i++ ) {
        int x1 = (i - 1) * width()    / points;
        int x2 = i * width() / points;
        p.drawLine( x1, h - int(systemLoad[i-1] * mult),
              x2, h - int(systemLoad[i] * mult) );
    }

    p.setPen( red ); //   设置红色的画笔，在整个画面上绘制应用程序负载曲线
    for ( int i = 1; i < points; i++ ) {
        int x1 = (i - 1) * width() / points;
        int x2 = i * width() / points;
        p.drawLine( x1, h - int(userLoad[i-1] * mult),
                    x2, h - int(userLoad[i] * mult) );
    }
#endif
}
```

函数 timeout 是槽函数，每当定时器超时，它被调用来从/proc/stat 中读出负载值，并将两次之间的负载值差除以时间差得到画在图上的负载值。该函数列出如下：

```
void Load::timeout()
{
#ifdef SYSINFO_GEEK_MODE
    int user;
    int usernice;
    int sys;
    int idle;
    FILE *fp;
    fp = fopen( "/proc/stat", "r" );//打开文件
    fscanf( fp, "cpu %d %d %d %d", &user, &usernice, &sys, &idle );
    user += usernice;
    fclose( fp );
    struct timeval now;
    gettimeofday( &now, 0 );
    int tdiff = now.tv_usec - last.tv_usec;  //与上一次的时间间隔
    tdiff += (now.tv_sec - last.tv_sec) * 1000000;
    tdiff /= 10000;
```

```
    int udiff = user - lastUser; //    与上一次的用户负载差
    int sdiff = sys - lastSys;    //与上一次的系统负载差
    if ( tdiff > 0 ) {  //负载计算
         double uload = (double)udiff / (double)tdiff;
         double sload = (double)sdiff / (double)tdiff;
         if ( !first ) {
            //负载点在图画上向前移动一个点
            for ( int i = 1; i < points; i++ ) {
                 userLoad[i-1] = userLoad[i];
                 systemLoad[i-1] = systemLoad[i];
            }
            userLoad[points-1] = uload;
            systemLoad[points-1] = sload;
            repaint( TRUE );//重画图
            double ml = 1.3;

            if ( maxLoad != ml ) {
                 maxLoad = ml;
                 update();
            }
         }

         last = now;
         lastUser = user;
         lastSys = sys;
         first = FALSE;
    } else if ( tdiff < 0 ) {
         last = now;
    }
#endif
}
```

7.5 内存信息

MemoryInfo 类窗口显示设备的内存信息，该类的构造函数创建 MemoryInfo 窗口中各窗口部件的对象和一个定时器。该定时器每隔一定时间从/proc/meminfo 文件中读取内存信息并更新窗口的显示，这个操作是由函数 updateData()完成的。MemoryInfo 类的窗口及各窗口部件类的位置如图 7-12 所示。

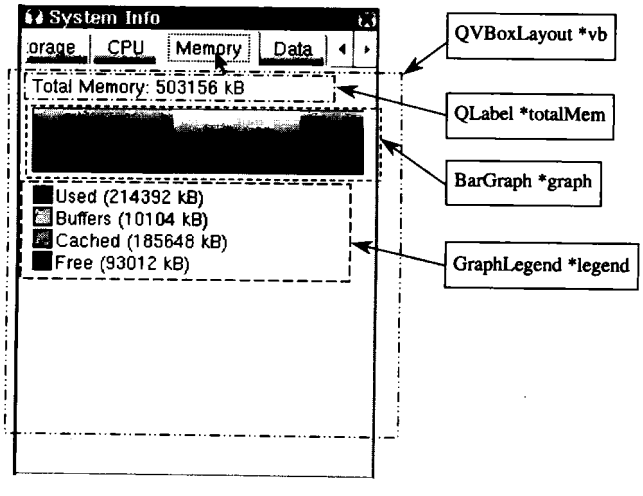


图 7-12 MemoryInfo 类窗口及各窗口部件类的位置

函数 updateData 列出如下：

```
void MemoryInfo::updateData()
{
   QFile file( "/proc/meminfo" );

   if ( file.open( IO_ReadOnly ) ) {
      QTextStream t( &file ); //字符流
      QString word;
     int total, memfree, buffers, cached;
      while ( ! t.atEnd() ) {  //从字符流中读出内存各项值
         t >> word;
         if ( word == "MemTotal:" ) {
            t >> total;
         } else if ( word == "MemFree:" ) {
            t >> memfree;
         } else if ( word == "Buffers:" ) {
            t >> buffers;
         } else if ( word == "Cached:" ) {
            t >> cached;
            break;
         }
      }

        int realUsed = total - ( buffers + cached + memfree );
        data->clear();

        // Geek mode
        data->addItem( tr("Used (%1 kB)").arg(realUsed), realUsed );
        data->addItem( tr("Buffers (%1 kB)").arg(buffers), buffers );
        data->addItem( tr("Cached (%1 kB)").arg(cached), cached );
        data->addItem( tr("Free (%1 kB)").arg(memfree), memfree );
        totalMem->setText( tr( "Total Memory: %1 kB" ).arg( total ) );
        ...
        graph->repaint( FALSE ); //重画 graph
        legend->update();   //更新 legend
   }
}
```

7.6 DataView 类

DataView 类计算出每类文档的大小，并用图形方式显示在窗口中。 DataView 类窗口如图 7-13 所示。由于这个类与上面的分析相似，所以不再赘述。

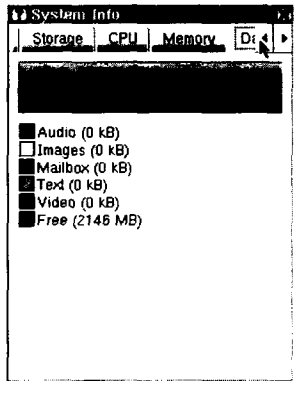


图 7-13 DataView 类窗口

第 8 章　Qtopia 编译及系统集成

嵌入平台的多样性使得 Qtopia 软件必须具有灵活性,要求 Qtopia 软件能根据具体嵌入设备进行软件定制和集成。并且因为 Qtopia 软件需要在 PC 机和嵌入设备上调试运行，所以要求 Qtopia 软件能在 PC 机的 x86 平台及嵌入设备平台（一般为 xscale 平台）编译，并且可生成 debug 与 release 版本。这些原因造成了 Qtopia 编译环境较复杂，选项较多。

本章说明 Qtopia 的定制与集成、.pro 项目文件的语法、由.pro 项目文件生成 Makefile 文件的过程，以及 Qtopia 的打包、安装和调试方法。

8.1　qmake 介绍

由于直接为不同的平台和编译器编写一个 Makefile 较难且容易出错，所以 Trolltech 公司创建了 qmake 工具来自动生成 Makefile。qmake 处理对编译器和平台的依赖性，以及 moc 和 uic 的连编规则。开发者创建一个纯文本的 *.pro 项目文件来指定其源代码文件，然后运行 qmake 生成适当的 Makefile。

在不同的平台上编译应用程序，需要设置 QMAKESPEC 环境变量。在 Windows 下使用 Microsoft Visual Studio ,要把 QMAKESPEC 环境变量设置为 win32-msvc;在 Solaris 上使用 gcc,把 QMAKESPEC 环境变量设置为 solaris-g++。

在 qt/mkspecs 中的每一个目录中都有一个包含平台和编译器特定信息的 qmake.conf 文件，这些设置适用于要使用 qmake 的任何项目，不要修改它。

8.1.1　qmake 编译过程

下面用一个简单的例子说明使用 qmake 命令通过.pro 文件生成 Makefile 的过程。

.pro 文件如下：

```
SOURCES = hello.cpp main.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

第 1 行指定了实现应用程序的源程序文件。另一种方法是使用 +=增加一个文件，它只是在已有的列表中添加新的文件，而不是替换整个列表。列出如下：

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

类似地，在 HEADERS 行中用来给这个应用程序增加一个头文件，列出如下：

```
HEADERS += hello.h
```

CONFIG 变量说明应用程序的配置信息，根据 CONFIG 配置信息，可判断连接哪些库等。CONFIG 行告诉 qmake 应用程序连接 qt 库，warn_on 部分告诉 qmake 把编译器设置为输出警告信息，release

部分告诉 qmake 应用程序被连编为一个发布的应用程序。

默认时，编译生成应用程序名被设置为和项目文件一样的名称，可以在项目文件中设置一个生成目标名。例如，如果不想使用 hello 作为目标名，可以在项目文件中设置目标名为 helloworld，列出如下：

```
TARGET = helloworld
```

在定义 .pro 文件后使用下面的命令生成 Makefile 文件：

```
qmake -o Makefile hello.pro
```

在生成 Makefile 文件后可以在 Makefile 所在目录下输入 make 或者 nmake 命令，计算机会开始编译连接应用程序。

应用程序的发布版本不包含任何调试符号或者其他调试信息。若需要应用程序的调试版本时，在项目文件的 CONFIG 变量中添加“ debug ”即可简单地实现。列出如下：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

在 .pro 文件中还可以判断平台来编写不同的代码，如为 Windows 平台添加的依赖平台的文件如下：

```
win32 {
   SOURCES += hello_win.cpp
}
```

8.1.2 .pro 文件的语法分析

qmake 使用的语法规则不多，为了便于理解，下面用一个实际使用的 .pro 文件为例来说明 .pro 文件中的语法。这个 test.pro 文件将各个 .cpp 及 .h 文件连接成一个应用程序，列出如下：

```
#qt 表示连接 qt 相关库，warn_on 表示打印警告信息，thread 表示这是一个多线程应用程序
CONFIG += qt warn_on thread merge_app

TEMPLATE = app #创建应用程序的 Makefile
#函数 include 把文件 default.cfg 的内容插到此处，函数返回值表示是否执行成功
include(./$$PRODUCT/default.cfg)

merge_app {
     DEFINES += TEST_MERGE_APPLICATIONS
}

#放置可执行程序目标的目录
DESTDIR = $(TEST_APP_BASE)/am/data
#可执行应用程序的名称，默认值为项目文件的名称
TARGET = am

#Java Session:
#应用程序中用到的预处理定义列表
DEFINES += JAVA_SESSION
DEFINES += AMRAD_DEBUG_MODE

#应用程序所需的头文件列表
HEADERS = \
     $(TEST_APP_BASE)/am/mainscreen/engine/AM_StartupInit.h \
     $(TEST_APP_BASE)/testsync/ui/usb_control.h
```

```
#应用程序所需的源文件列表
SOURCES = \
     $(TEST_APP_BASE)/testsync/ui/usb_control.c

#如果 MAKE_FTR_LED 存在于 DEFINES 变量的值的列表中
contains(DEFINES, MAKE_FTR_LED){
HEADERS += \
       $(TEST_APP_BASE)/am/mainscreen/engine/AM_LED_EventManager.h
SOURCES += \
       $(TEST_APP_BASE)/am/mainscreen/engine/AM_LED_EventManager.cpp
}

#应用程序的 include 路径的列表
INCLUDEPATH += \
     $(TEST_APP_BASE)/include \
     $(TEST_APP_BASE)/utility/inc \
       /vobs/test_api/code/utility/inc
}

#unix表示 qmake将在 unix模式运行，这种模式下将使用 unix 文件命名和路径转换
#在 unix 下这是默认模式
#LIBS变量包括连接进项目的库的链表，如果使用了 -L/-l 的 unix 转换
#那么可以在交叉编译方式下任意使用它们
#qmake将会在 Windows 上正确使用这些库（即传递库的全路径给连接器）
#qmake在 -L 所指路径下找到 -l 所指的库
unix:LIBS += -L$(TEST_APP_BASE)/lib
unix:LIBS += -ltestdb

#如果 CONFIG 变量链表中含有 debug，即是调试版本
contains(CONFIG, debug) {
     contains(QMAKE_LINK, g++) {  #含有 g++, 即 x86 下的调试版本
        DEFINES += AM_X86
        unix:LIBS += -ltest-x86-d
        unix:LIBS += -ltestasprofile-x86-d
    }

contains(QMAKE_LINK, arm-linux-g++) {#含有 arm-linux-g++, 即 arm cpu 下的调试版本
     unix:LIBS += -ltest-xscale-d
     unix:LIBS += -ltestringtone-play-xscale-d
     unix:LIBS += -ltestasprofile-xscale-d
}

#CONFIG 变量中含有 relese，表示是发布版本
contains(CONFIG, release) {
     contains(QMAKE_LINK, g++) { #含有 g++, 即 x86 下的发布版本
        DEFINES += AM_X86

        unix:LIBS += -ltestopenwindow-x86-r
         unix:LIBS += -lteststaticidle-x86-r
     }

    #含有 arm-linux-g++, 即 arm cpu 下的发布版本
     contains(QMAKE_LINK, arm-linux-g++) {
        unix:LIBS += -ltestam-xscale-r
        unix:LIBS += -ltestopenwindow-xscale-r
     }
}

# 这是国际化语言文件
TRANSLATIONS = \
     $(TEST_APP_BASE)/am/am_zh-cn.ts
```

一些语法说明如下。

（1）TEMPLATE

TEMPLATE 变量包括产生项目时使用的模板名字，它允许的值说明如下。

- app：默认值，为编译应用程序创建一个 Makefile。
- lib ：为编译库创建一个 Makefile。
- subdirs ：为在子目录中编译目录创建一个 Makefile。
- vcapp：win32仅创建一个应用程序项目文件。
- vclib：win32仅创建一个库项目文件。

（2）*=操作符

*=操作符仅仅在一个值不存在于一个变量值的列表中时添加该值，使用方法如下：

```
DEFINES *= QT_DLL
```

只用在 QT_DLL 没有定义在预处理定义的列表中时，它才会被添加。

（3）~=操作符

~=操作符将会替换任何与指定值的正则表达式匹配的任何值，使用方法如下：

```
DEFINES ~= s/QT_[DT].+/QT
```

这将会用 QT 来替代任何以 QT_D 或 QT_T 开头的变量中的 QT_D或 QT_T。

（4）常用的系统变量

常使用的系统变量说明如下。

- LIBS：应用程序中连接的库及连接选项。
- HEADERS：应用程序中的所有头文件的列表。
- SOURCES 应用程序中的所有源文件的列表。
- FORMS：应用程序中的所有 .ui 文件（由 Qt 设计器生成）的列表。
- TARGET ：可执行应用程序的名称，默认值为项目文件的名称。如果需要扩展名，则自动添加。
- DESTDIR：放置可执行程序目标的目录。
- DEFINES：应用程序所需的额外预处理程序定义的列表。
- INCLUDEPATH 应用程序所需的额外包含路径的列表。
- DEPENDPATH：应用程序所依赖的搜索路径。

8.2 Qtopia 编译方法

8.2.1 Qtopia 编译步骤

你首先需要从 www.trolltech.com 网站上下载 Qt/X-3.3.x、Qt/X11 2.3.2、Qt-Embedded 2 3.10 和 Qtopia 2.1.1 版本并在 Linux 下解压缩，然后使用 export 设置环境变量指向源代码目录。环境变量说明如下。

（1）QT2DIR：指 向 Qt-X11 2.3.2 源代码所在的目录。

（2）DQTDIR：指 向 Qt-X11 3.3 源代码所在的目录。

（3）QTEDIR：指 向 Qt-Embedded 2.3.10 源代码所在目录。

（4）QPEDIR： 指向 Qtopia 源代码所在目录。

Qtopia 提供了一个编译脚本文件 build_qtopia 简化编译，但如果编译不成功，则很难找出问题，因此建议使用后面的方法一步步地编译。

直接运行 Qtopia 提供的脚本进行编译的方法如下：

```
$QPEDIR/scripts/build_qtopia
```

如果需要保存编译输出的信息，可使用如下命令编译：

```
$QPEDIR/scripts/build_qtopia > log.txt 2>&1
```

Qtopia 的编译分为在 PC 机上 qvfb 环境下运行的版本和在 PDA 设备上下载运行的版本两种，在 PC 机上 qvfb 环境下运行的版本使用基于 x86 处理器的编译器；在 PDA 设备上下载运行的版本使用基于 Arm 处理器的交叉编译器。

在 PC机上 qvfb环境下运行的版本从源代码逐步编译的方法如下。

（1） 编译 Qt 3.3及工具

编译 Qt 3.3 的目的是为了在 PC 机上运行 qvfb，如果可运行 qvfb，则不必编译它。编译 Qt 3.3及工具的步骤如下：

```
cd $DQTDIR
export QTDIR=$DQTDIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
echo yes | ./configure -fast -thread
make
make install
```

(2)编 译 Qt 2.3.2

编译 Qt 2.3.2的步骤如下：

```
cd $QT2DIR
export TMAKEPATH=$TMAKEDIR/lib/linux-g++
export QTDIR=$QT2DIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
./configure -no-xft
make
mkdir $QTEDIR/bin
cp bin/uic $OTEDIR/bin/
```

(3)编 译 Qt/Embedded

编译 Qt/Embedded 的步骤如下：

```
export LD_LIBRARY_PATH=$QT2DIR/lib:$LD_LIBRARY_PATH
cp $QT2DIR/bin/qvfb $QTEDIR/bin/
cp $QT2DIR/bin/uic $QTEDIR/bin/
cd $QTEDIR
export TMAKEPATH=$TMAKEDIR/lib/qws/linux-x86-g++
export QTDIR=$QTEDIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
cp $QPEDIR/src/qt/qconfig-qpe.h src/tools/
./configure -system-jpeg -no-xft -qconfig qpe -keypad-mode -qvfb -depths 4,8,16,32 [extra options:
see below]
make
```

(4)编 译 Qtopia

由于 Qtopia 开放源代码只有 PDA 版本，因此这里只说明 PDA 版本的编译步骤，在 PC 机上 qvfb 下运行的 Qtopia 编译步骤如下：

```
cd $QPEDIR
export QTDIR=$QTEDIR
export PATH=$QPEDIR/bin:$PATH
```

```
./configure -edition pda -no-qtopiadesktop -without-libamr -without-libffmpeg #
```
由于这两个库编译出错，故去掉这两个库

```
make
make install
```

在 PDA 设备上下载运行版本的编译步骤非常类似于在 PC 机上运行的版本。在 PDA 设备上运行的程序不需要 qvfb，因此不需要编译 Qt 2.3.2 和 Qt 3.3。

在编译过程中使用 configure 配置源代码时有一个选项 -platform 用来选择 custom-<platform-spec>.h 文件，设备硬件特定的宏定义会列出在这个 custom 文件中，相应的 custom-<platform-spec>.cpp 文件提供需要的硬件特定函数。

在 PDA 设备上下载运行的版本的编译步骤如下。

（1）拷贝 uic

方法如下：

```
mkdir $QTEDIR/bin/
cp $QT2DIR/bin/uic $QTEDIR/bin/
```

（2）交叉编译 Qt/Embedded

交叉编译 Qt/Embedded 生成基于 Arm 处理器在设备上运行的版本，编译方法如下：

```
cd $QTEDIR
export TMAKEPATH=$TMAKEDIR/lib/qws/linux-arm-g++
export QTDIR=$QTEDIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
cp $QPEDIR/src/qt/qconfig-qpe.h src/tools/
./configure -xplatform linux-arm-g++ -system-jpeg -no-xft -qconfig qpe -keypad-mode -no-qvfb -depths
4,8,16,32
make
```

（3）交叉编译 Qtopia

交叉编译 PDA 版本的 Qtopia 的方法如下：

```
cd $QPEDIR
export QTDIR=$QTEDIR
export PATH=$QPEDIR/bin:$PATH
./configure -xplatform linux-arm-g++ -edition pda -no-qtopiadesktop
make
make install
```

将编译好的软件进行打包处理的方法如下：

```
cd $QPEDIR
make packages
```

8.2.2 Qtopia Desktop 编译步骤

Qtopia Desktop 是在 PC 机上运行并与设备上 Qtopia 进行同步操作的应用程序，它有 Windows、Linux 及 Mac Os 等操作系统上的版本。本节只说明在 Linux 版本的编译步骤，它与 Qtopia 的 x86 版本编译类似，但 Qt 3.3.0 版本需要商业版本。如果用户不需要修改 Qtopia Desktop 应用程序，可以下载 rpm 安装版本直接安装。

Qt 3.3.0 的编译步骤如下：

```
tar xfz qt-x11-commercial-3.3.0.tar.gz
```

```
cd qt-x11-commercial-3.3.0
export QTDIR=$PWD
export PATH=$QTDIR/bin:$PATH
./configure -thread
make
cd ..
```

Qtopia Desktop 的编译步骤如下：

```
tar xfz qtopia-source-2.1.1.tar.gz
cd qtopia-source-2.1.1
export QPEDIR=$PWD
export DQTDIR=$QTDIR
export PATH=$QPEDIR/bin:$DQTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QPEDIR/lib:$LD_LIBRARY_PATH
/configure -no-qtopia
make
rm -rf dimage
make install
```

Qtopia Desktop 现在被安装在$QPEDIR/dimage/opt/Qtopia 目录下。

8.2.3 编译一个应用程序

可以在 Qt 适用的任何平台上利用 Qt 开发应用程序，但为了使应用程序适合于嵌入设备的屏幕和其他限制，建议使用 Qtopia SDK。Qtopia 应用程序可以在 Linux 平台 PC 机 X11 上使用 Qt 虚拟帧缓冲 qvfb 仿真运行，通过目标机器对应的交叉编译软件，可以将源代码编译成目标机器上运行的应用程序。

一个运行在 Linux PC 机上的样例应用程序的编译步骤如下。

（1）检查 Qtopia SDK 环境变量的设置，该变量设置如下：

```
export QPEDIR=/opt/Qtopia
export QTDIR=$QPEDIR
export DQTDIR=$QPEDIR
export PATH=$QPEDIR/bin:$PATH
export QMAKESPEC=qws/linux-generic-g++
export LD_LIBRARY_PATH=$QPEDIR/lib:$LD_LIBRARY_PATH
```

（2）将 Qtopia SDK 中样例应用程序源代码拷贝到用户自己定义的 somewhere 目录下：

```
cd somewhere
cp -r $QPEDIR/examples/application .
chmod +w application/*
cd application
```

（3）使用 qmake 产生 Makefile 文件：

```
qmake -o Makefile application.pro
```

（4）编译样例如下：

```
make
```

（5）安装样例（需要 root 权限）如下：

```
su #  安装需要 root 权限
make install INSTALL_ROOT=$QPEDIR
```

```
exit #  不再需要 root 权限
```

在 Linux PC 机上运行样例应用程序的步骤如下。

（1）后台运行 Qt/X11 虚拟帧缓冲工具

在 PC 机的终端窗口输入以下命令：

```
qvfb &
```

应用程序可以通过 qvfb 运行在 Linux PC 机上，可以使用类似下面的命令为 qvfb 设置一个如同设备一样的外形窗口：

```
qvfb -skin $QPEDIR/bin/SmartPhone.skin &
```

（2）运行 Qtopia 环境来运行应用程序

在 PC 机的终端窗口输入以下命令：

```
./qpe
```

样例应用程序应该出现在 Qt/X11 虚拟帧缓冲窗口中的应用程序表中。

（3）直接运行应用程序

在 PC 机的终端窗口输入以下命令：

```
./application -qws
```

还可以将应用程序打成 RPM 包来发布，方法如下：

```
mkipks -rpm -arch i386 example.control
```

卸载已安装的样例程序方法（需要 root 权限）如下：

```
su #安装需要 root 权限
make uninstall INSTALL_ROOT=$QPEDIR
exit #不再需要 root 权限
```

编译一个在 SHARP SL5000 嵌入设备上运行的应用程序的方法类似于在 PC 机上编译的方法，样例假设已安装交叉编译器在目录/usr/local/arm 中，编译步骤如下。

（1）检查 SHARP SL5000 编译环境。

```
export QPEDIR=/opt/Qtopia/arm
export QTDIR=$QPEDIR
export DQTDIR=/opt/Qtopia
export QMAKESPEC=qws/linux-sharp-g++
export PATH=$QPEDIR/bin:/usr/local/arm/bin:$PATH
```

（2）重新生成 Makefile，因为 $QMAKESPEC 已变化。

```
make clean
qmake -o Makefile application.pro
```

（3）编译。

```
make
```

（4）安装（注意现在安装进/opt/Qtopia/arm/ 目录）。

```
su #安装需要 root 权限
```

```
make install INSTALL_ROOT=$QPEDIR
exit #不再需要 root 权限
```

（5）为目标设备创建和安装 Qtopia 的 image 文件。

当为目标设备配置 Qtopia 时，$QPEDIR/image 是即将创建的 image 的默认目录。在编译安装后的代码放在 image 目录下，将该目录下所有文件使用 tar 命令归档打包，然后在目标设备上从归档文件中提取 image 文件。

在编译的主机上使用 tar 命令将当前目录下进行归档（即将目录及文件放在一个文件中），归档后的文件名为 qtopiaimage.tar。命令如下：

```
cd $QPEDIR/image
tar -cf   $HOME/qtopiaimage.tar
```

在目标设备上使用 NFS 文件系统挂接主机上的文件目录，命令如下：

```
mount -t nfs myhost:/home/myuser /mnt/tmp
```

在目标设备上从归档文件 qtopiaimage.tar 中提取 image 文件，命令如下：

```
cd /
tar xf /mnt/tmp/qtopiaimage.tar
```

8.2.4 使用 NFS 运行应用程序

Qtopia 可运行在仿真环境和嵌入设备上，运行于 PC 机上仿真环境中可方便开发与调试。应用程序运行在真实设备上可检验代码的执行效率，运行在设备上的应用程序需要通过交叉编译得到对应设备的应用程序二进制代码。

使用 NFS 挂接分区来在一个设备上运行 Qtopia 的基本思想是设备有一个网络与 PC 机连接并访问 PC 机上的 Qtopia 二进制代码，这个二进制代码必须是交叉编译器生成的对应设备的二进代码。通过连接设备的一个终端来启 /停应用程序，这种方法对于在真实设备上开发和调试是方便的。二进制代码能被快速生成，并可立即启动，从而省略了缓慢的二进制程序下载进设备的时间。

使用 NFS 来运行应用程序的步骤如下。

（1）配置设备

使用命令行工具对设备进行网络配置并用 ifconfig 设定 IP 地址。

（2）确信 Linux PC 机上运行 nfs 服务器

确信 nfs 服务器已在 PC 机上运行，使用下面 ps 命令，可以看到 nfsd 进程的存在：

```
tester@desk:~> ps aux | grep nfs
root     10661  0.0  0.0     0    0 ?        SW   16:20   0:00 [nfsd]
root     10662  0.0  0.0     0    0 ?        SW   16:20   0:00 [nfsd]
tester   10554  0.0  0.1  3540  564 pts/8    S    16:12   0:00 grep nfs
```

（3）指示 NFS 输出的目录

以 root 身份在文件/etc/exports 中加入下面一行：

```
your-qpe-source-dir/image/opt/Qtopia/ *(ro)
```

这一行的意思是暴露 your-qpe-source-dir/image/opt/Qtopia/bin directory 及其所有的子目录，任何其他机器可以以仅读方式访问这个目录。

使用下面的命令告诉 NFS（以 root 身份）配置发生变化：

```
exportfs -a
```

这时，在 your-qpe-source-dir/image/opt/Qtopia/bin 目录下的 Qtopia 二进制代码应能从网络上被访问到。

（4）连接设备

通过 telnet 或 ssh（使用设备 IP 地址）注册连接到设备上，以 root 身份登录并在设备上创建能连接到 PC 机文件系统的目录。创建目录命令如下：

```
mkdir -p /opt/Qtopia
```

使用下面命令将 PC 机文件系统挂接到设备中的目录上：

```
mount desk:your-qpe-source-dir/image/opt/Qtopia -orsize=8192,wsize=8192 /opt/Qtopia
```

这一行中 desk 为 NFS 服务器所在的 PC 机器名，这样即可运行通过 NFS 在设备上运行 NFS 服务器上 的应用程序。

8.2.5 应用程序调试

应用程序的调试方法有 3 种，即打印信息、NFS 及打印信息和 DDD 调试 3 种方式，分别说明如下。

第 1 种是在设备上运行 Qtopia，通过打印语句 printf 或 qDebug 函数在终端上打印变量信息来调试应用程序。这个终端可以在 PC 机通过 telnet 命令登录到设备上，或者直接使用设备上的终端窗口。

第 2 种方式是使用 NFS 方式使应用程序在 PC 机上运行，打印变量信息同样显示在终端窗口中，显示方法与第 1 种相同。与第 1 种方式相比，优越性在于由于应用程序在台式机上运行，每次修改不必将应用程序烧录到设备上。这样，省略了频繁使用烧录设备的麻烦。

第 3 种方式是将应用程序编译成 x86 版本，在 PC 机上用 DDD 调试工具进行调试。下面对这种方式进行详细说明。

对于 Qtopia 应用程序来说，可以把应用程序编译成 x86 下的调试版本，即在 Qtopia 编译执行 configure 脚本时带上 -debug 选项。生成的应用程序可以在 PC 机上使用 qvfb 来运行。

对应用程序的调试使用 Linux 的 DDD 图形调试工具较直观，下面以 clock 应用程序为例说明如何使用 DDD 工具来调试应用程序。

在 PC 机终端窗口中输入的命令如下：

```
cd $QTOPIA_PATH/bin
qvfb&
ddd clock
```

此时，在 DDD 工具窗口中出现应用程序源代码，应用程序停止运行。用户在应用程序源代码中需要设置断点处设置断点，然后在 DDD 命令窗口中输入 run –qws。此时应用程序继续执行，并停止在断点处。这时，用户可单步执行程序，或者查看应用程序执行过程中的各种参数，如变量值及寄存器值等。DDD 工具调试 clock 应用程序的示例如图 8-1 所示。

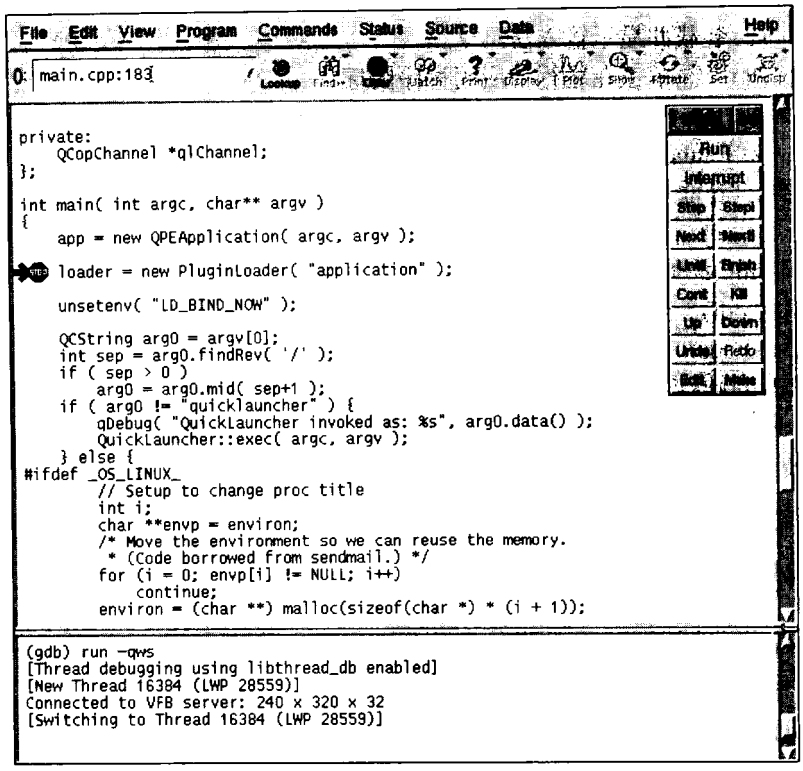


图 8-1 DDD 工具调试 clock 应用程序的示例

8.3 为设备定制 Qtopia

如何在一个给定的设备上安装 Qtopia 将由设备的功能来决定，根据不同设备，用户需要对 Qtopia 软件进行定制，选择安装适合于设备的软件组件。

8.3.1 定制 Qtopia 启动器用户界面

Qtopia 提供了 PDA 设备通用的应用程序启动器（Launcher）界面，它用 Launcher 类描述。但有时这个 PDA 的默认 Launcher 并不适合指定的设备，需要用户定制 Launcher。Qtopia 提供了 ServerInterface 类对 Launcher 的用户界面接口进行规范，ServerInterface 界面接口类提供了维护 Qtopia 应用程序及文档模型相适应的所有需要的界面函数，用户编写的定制 Launcher 必须提供这些界面函数的实现。应用程序启动器用户界面（Launcher User Interface）是 Qtopia 服务器（$QPEDIR/src/server）的一部分，Qtopia 服务器提供了诸如同步和文件系统管理的功能。

（1）定制 Launcher 的方法

编写应用程序启动器界面的第 1 步是从 ServerInterface 派生类，这个派生类提供了在 Qtopia 设备上显示可用应用程序和文档的必要函数接口。函数 createGUI()和 destroyGUI(必须被用来创建和显示 Launcher 及其他用户界面部件。

ServerInterface 类提供的大多函数被用来增加或删除应用程序及文档，这些功能是必需的。

（2）其他用户界面组件

Launcher 本身也需要提供一些系统组件，如输入法及状态显示等。在默认的 Qtopia 用户界面中有多个类可在定制的界面中使用，大多数用户界面将使用这些类来确保系统可用。在定制的 Launcher 中

必须使用到的类说明如下。

（1）InputMethods：装载并显示输入法。

（2）RunningAppBar ：显示正在运行的应用程序图标。

（3）SysTray：装载并显示小应用程序。

（4）StartMenu ：为 Launcher 的 Tab 提供菜单和必要的插件。

提供了这些组件的界面将支持默认的 Qtopia 应用程序启动器大部分功能， Qtopia 库还假定 QPE/Taskbar 信道是可用的，并运行下列消息处理函数。

（1）message(QString)：显示一个状态消息。

（2）hideInputMethod()：隐藏所有可见的输入法。

（3）showInputMethod()：显示当前选择的输入法。

（4）showInputMethod(QString)：显示特定的输入法。

（5）reloadInputMethods() ：释放并重装载输入法插件。

（6）reloadApplets() 释放并重装载小应用程序插件。

默认的 Qtopia 程序启动器界面使用 TaskBar 类来包裹上面这些功能函数。

8.3.2 设备硬件因素

在定制和安装 Qtopia 到一个特定设备上时，必须考虑多个因素，下面是一些典型的因素说明。

如果设备有一个触摸屏，Qtopia 刻度校准必须被激活。

如果设备耗尽电池，Qtopia 需要调用 APM 函数进行电源管理。

如果设备有特殊的硬件按钮，它们需要被映射到应用程序并与用户界面中显示的图标联系起来。

如果设备有一个报警用的蜂鸣器及其他的喇叭， Qtopia 将需要知道如何使用这些硬件设备并用相关的事件激活它。

如果设备有一个带有背景或前景灯的 LCD 屏，Qtopia 需要知道如何调节其亮度。

如果设备有任何特殊的 LED 用来指示使用者的状态，Qtopia 需要知道如何设置 LED。

如果设备有一个数字相机，则必须提供一个 Video4Linux 内核驱动程序，以便 Qtopia 应用程序能够使用它。

如果设备有或支持手机硬件，则必须提供一个支持 GSM AT 命令的串行设备，以至于 Qtopia Phone 能使用它。

如果设备支持声音，音量应被锁定到最大需要。

定制这些设备特定的软件组件是 Qtopia 很重要的工作，这些定制包括在一些小文件中，下面将会说明这些文件。

在运行 confugure 脚本时，-platform 选项用来选择哪个 custom-<platform-spec>.h 文件将在编译 Qtopia 时使用。在一个 custom.h 文件中与硬件相关的宏可能被定义来为特定的硬件定制 Qtopia，相关联的 custom-<platform-spec>.cpp 文件将被编译并连接以提供要求的定制函数。

与硬件相关的特定宏定义及其说明如表 8-1 所示。

表 8-1 与硬件相关的特定宏定义及其说明

宏定义名	宏定义说明
QPE_USE_MALLOC_FOR_NEW	用#define 定义这个宏，内存分配将使用 malloc()代替默认的 C++的 new 函数。在某些情况下，mallock()将执行更快
QPE_NEED_CALIBRATION	如果有一个触摸屏需要刻度校准，用#define 定义这个宏
QPE_OWNAPM	如果这个设备存在 APM 设置，用#define 定义这个宏。这个 APM 设置必须首先由 Qtopia 通过/dev/apm_bios 的 ioctl 函数取消

续表

宏定义名	宏定义说明
QPE_HAVE_TOGGLELIGHT	如果设置存在一个用于开/关 LCD 背景灯的硬件按钮，用#define 定义这个宏必须应用
QPE_HAVE_MEMALERTER	如果设备需要 Qtopia 在低内存状态时为用户报警，用#define 定义这个宏。一个为 initMemalerter()的函数
QPE_MEMALERTER_IMPL	如果需要使用带有 MEMALERTER 功能（即包括 initMemalerter()函数）的任何函数，用#define 定义这个宏
QPE_INITIAL_NUMLOCK_STATE	如果使用任何设备特定的方法初始化 numlock 状态，用#define 定义这个宏
QPE_ARCHITECTURE	如果使用了含有生产厂商和惟一鉴别设备模型的字符串，如“Manufacture/Device 1000”，用#define 定义这个宏
QPE_DEFAULT_TODAY_MODE	用#define 定义这个宏，告诉 Qtopia 默认下它是否应该每天运行 today 应用程序或从不运行
QPE_FONT_HEIGHT_TO_ICONSIZE(x)	用#define 定义这个宏，提供了从一个字体高度到一个图标尺寸的映射，单位为像数点
QPE_SYSTEM_SYSFILEMONITOR	如果设备有可移去的存储卡的 HotPlug ，系统是 Qtopia 能监控的，用#define 定义这个宏
CUSTOM_SOUND_INIT	用#define 定义这个宏，表示定义了一个函数来初始化任何定制的声音设备，如蜂鸣器
CUSTOM_SOUND_IMPL	用#define 定义这个宏，表示定义了被其他 CUSTOM_SOUND 宏使用时需要的任何定制函数
CUSTOM_SOUND_ALARM	用#define 定义这个宏，表示系统使用了设备特定的代码发出报警声
CUSTOM_LEDS(led, status)	用#define 定义这个宏，表示如果 LED 可用，系统能设置 LED 到一个给定的状态。参数 led 表示是哪个 LED，status 是设备特定含义的状态整数
CUSTOM_SOUND_TOUCH(press)	用#define 定义这个宏，表示如果屏幕可用，一个给定的声音将与触屏动作相关联
CUSTOM_SOUND_KEYCLICK(k, p, r)	用#define 定义这个宏，表示一个给定的声音与键盘按键相关联

除了上面这些宏定义外，表 8-2 中所示的 3 个函数必须在 custom-<platform-spec>.cpp 文件中实现。

表 8-2 硬件特定函数及其功能说明

函数名称	功能说明
int qpe_sysBrightnessSteps()	这个函数被 Qtopia 调用来查询设备的 LCD 背景/前景灯亮度增加的级数（即分几级达最大亮度）
void qpe_setBrightness(int)	这个函数必须与设备特定的代码一起使用来设置背景/前景灯的亮度级数
void PowerStatusManager::getStatus()	这个函数被用来查询电源状态，在 qtopia/power.h 中的 PowerStatus 类定义了电源状态

在 src/libraries/qtopia/ 目录下面可找到 custom-*.*形式的文件，例如对于 cassiopia 设备来说，custom-cassiopia-g++.h 文件内容列出如下：

```
#define QPE_NEED_CALIBRATION
#define QPE_USE_MALLOC_FOR_NEW
#define QPE_ARCHITECTURE "Casio/Cassiopeia"
#define QPE_DEFAULT_TODAY_MODE "Daily"
```

对于 cassiopia 设备来说，custom-cassiopia-g++.cpp 文件的内容列出如下：

```
int qpe_sysBrightnessSteps()  //设置亮度的级数
 {
    return 255;
}

void qpe_setBrightness(int bright)  // 设置 bright 亮度，它需要根据具体设备来实现
{

}

void PowerStatusManager::getStatus() //得到电源状态
```

```
{
    int ac, bs, bf, pc, sec;
    if ( getProcApmStatus( ac, bs, bf, pc, sec ) ) {  //从/proc/Apm 中读取电源状态
        ps->percentRemain = pc;
        ps->secsRemain = sec;
        ps->percentAccurate = TRUE;
    } else {
        ps->percentRemain = 100;
        ps->secsRemain = -1;
        ps->percentAccurate = FALSE;
    }
}
```

8.3.3 不安装及不支持的组件

有大量的与 Qtopia 一起提供的应用程序不需要安装到产品设备中，这些软件组件及其说明如表 8-3 所示。

表 8-3 不需要安装到产品设备中的软件组件及其说明

软件组件名称	使用说明
Rotation Setting（旋转设置）	提供了测试与验证 Qtopia 在不同方向显示的能力，如果设备使用了变形显示驱动程序，则在开发期间可能是有用的
Shutdown Setting（关机设置）	提供一个开发工具，在产品设备上不是必需的，它能通过删除 apps/Settings/quit.desktop 文件而被移去。由于在大多设备上有一个物理电源按钮来开关电源，因此用户不期望手动地在软件上开启运行或关闭 Qtopia
screensize applet（屏幕尺寸小应用程序）	允许在开发期间为了快速测试不同屏幕大小而改变最大窗口尺寸，它仅是开发工具，最终用户并不需要这个功能
Terminal（终端）	提供开发工具，对于比较熟悉 Linux 的高级用户来说，把它提供作为一个独立可安装的包比较合理。然而大多数最终用户会认为其无用，因此建议这个组件在最终产品上不要包括进去
File Manager（文件管理器）	作为开发工具用，Qtopia 仅使用 Qtopia 的文档模型完成文档管理。File Browser 应用程序能让用户从设备上删除或增加文件，但它破坏了 Qtopia 的文档模型。为了保护系统文件和设置以及一些用户关键数据，我们不建议使用这个应用程序
VNC Viewer（VNC 客户端）	不支持第三方应用程序/端口，它们被用来验证 Qtopia 的扩展性

8.3.4 安全模式需要的插件

如果 Qtopia 连续崩溃两次，设备将进入“ Safe Mode”安全模式。这时插件将不被装载，以免插件引起崩溃。在安全模式的某些情况下，为了设备的正常操作，有些插件是需要的，这些插件能被定义在 $QPEDIR/plugins/<type>/.directory 文件中。例如，如果 simple8 字体的解码器总是需要的，可如下设置：

```
[Desktop Entry]
Name=Text Codecs
Comment=Provides support for various international languages.
Required=simple8
Apply=QPE/System restart()
```

8.3.5 电源管理

当在有电源管理的设备上集成 Qtopia 时，很多情况需要考虑，这些情况列出如下。

（1）Qtopia 控制电源状态，如使用触笔和按键输入时点亮屏幕。

（2）系统 Power 键仅发送一个键值给 Qtopia，但它并未执行实际的挂起操作。

（3）Qtopia 支持多种电源关闭级别，默认是 On、Light Dim、LightOff 和 Suspend，其他附加级别需要通过增加少量代码实现。

（4）Qtopia 调用平台特定的函数来设置屏幕的亮度级别并关闭屏幕。

（5）Qtopia 运行 apm 命令挂起电源。

（6）系统在后台进程依赖于 RTC 报警来从电源挂起状态中恢复运行。

（7）系统利用后台进程写 RTC 时钟（因为一些系统仅一个进程可处理 RTC）。

Qtopia 的电源管理通过 Qt/embedded 中继承于 QWSServer 的类 QWSScreenSaver 来处理，Qtopia 提供了从 QWSScreenSaver 扩展来的 QPEScreenSaver 类。qpeapplication.cpp 使用了 QPEScreenSaver 类提供了 3 种级别的屏幕保护，列出如下。

（1）0：使显示灯变暗。

（2）1：关闭显示灯。

（3）2：依赖于 Qtopia 版本。

Qtopia 调用平台特定的函数 qpe_setBrightness(int)来调节显示的灯光，级别 2 依赖于 Qtopia 版本。PDA 版本将使用 APM 后台进程，并挂起执行的应用程序，而 phone 版本则关闭应用程序并返回到 home 屏。

QWSServer 提供了下面的接口函数，允许用户设计屏幕保护的操作：

```
void setScreenSaver ( QWSScreenSaver *
void setScreenSaverIntervals ( int * ms
void screenSaverActivate ( bool )
```

系统后台进程依赖于 RTC 的报警从挂起状态恢复，系统后台进程被用来写 RTC 时钟（因为某些系统仅一个进程能处理 RTC）。

（1）PDA Edition 版本

Qtopia 运行 apm 命令挂起，系统的 AT 后台依赖于 RTC 报警来从挂起状态恢复并被用来写 RTC 时钟（因为某些系统仅一个进程能处理 RTC）。

系统的 Power 键或第 2 级的屏幕保护仅发送一个键值（Qt::KeyF34）到 Qtopia，它并未执行实际的挂起操作并触发 ServerApplication::togglePower() 函数的运行。电源挂起的操作命令如下：

```
system("apm --suspend")
```

一旦上面的命令被执行，则挂起所有的 CPU 功能。一旦一个 Wake up 事件来自 apmd 后台，将恢复正常运行。

（2）Phone Edition 版本

在 Phone Edition 版本中，挂起操作未被激活，因为命令 apm –suspend 将挂起 CPU 并关闭设备。这个行为对于 PDA 来说是可接受的，因为它们不必监听进来的电话；对于 Phone Edition 版本来说，qpe server 必须运行以接听电话。如果 Qtopia 关闭电源，则设备将停止工作。我们假定 Qtopia Phone 不这样做（如果这样做，Linux 确保通过 modem 活动能恢复运行）。Qtopia Phone 不是挂起设备，而是在 qpe server 正在显示主屏幕时降低 CPU 的频率到最小。在第 0 个级和第 1 个级屏幕保护级别（与 PDA edition 版本一样）. Qtopia Phone 自动关闭应用程序并返回到主屏幕。背景灯活动（如执行 time/date 更新和电源监视器）被降低，以确保 CPU 消耗保持在最低的水平。

8.3.6 可移去的存储卡

外部设备，如 SD 和 CF 卡在插入时将被扫描，并且支持的文档将被映射到 Qtopia 的文档视图中。

当 Qtopia 库通过 QPE/Card QCOP 通道收到下面的消息时扫描被触发。

（1）mtabChanged()：当/etc/mtab 已变化时发送消息。

（2）stabChanged()：当/var/run/stab,/var/state/pcmcia/stab,或/var/lib/pcmcia/stab 已变化时发送消息。

如果宏定义 QPE_SYSTEM_SYSFILEMONITOR 被定义，则与系统文件监控相关的软件会被编译进 Qtopia，Qtopia 系统会产生这些消息（通过使用系统 HotPlug 脚本中的 qcop 命令）。

当 Qtopia 扫描外部设备如 SD 和 CF 卡时，可以很好地识别外部卡。如果由于特定设备的原因，自动检测不足以识别外部设备时，可为在/etc/mtab 中的每个相关设备在/opt/Qtopia/etc/default/Storage.conf 文件中提供一个组（group）进行配置描述。设备的格式如下：

```
[/dev/hda1]
Name = Internal Storage
Removable = 0
[/dev/sda1]
Name = Card
Removable = 1
```

使用上面这个描述，/dev/hda1 将是内部的存储设备。并且在/dev/sda1 挂接时（如通过 Linux 热插拔系统），一个名为“ Card”存储位置对于用户来说是可用的，任何文档将出现在文档页中。

8.3.7 I18N

英文中的单词国际化（Internationalization）缩写为“I18N”， 它是取首尾两个字母，而中间字母个数为 18个，故称为“I18N”。Qtopia 所有的文件格式使用 UTF8字符编码，系统函数也使用 UTF8 编码输出。例如，$LANG 设置为 “ja”，strftime()必须返回UTF8 编码的字符串。依赖于操作系统，可能要求“ ja”是“ ja_JP.utf8 ”本地化的别称。

注意 Qtopia 期望所有的文件系统使用 UTF8 编码，为此需要注意/etc/mtab 和 /usr/share/locale，特别需要注意可移去的媒体。

8.3.8 访问权限和仅读文件系统

Qtopia 可作为任何用户运行，用户有读与写下面文件的权限。

（1）/dev/fb0：Frame Buffer

（2）/etc/ppp/...：PPP 配置选项。

（3）/etc/pcmcia/...：LAN/wireless 网络选项。

（4）/opt/Qtopia/...：安装附加的软件。

如果使用诸如 cramfs仅读文件系统，那么应创建一个符号链接到一个可写目录以便上述目录是可写的。例如， /opt/Qtopia/apps/Applications/calculator.desktop 是 /opt/Qtopia-ROM/apps/Applications/calculator.desktop 的符号链接。

作为 $QPEDIR 指定目录的替换者，大多数 Qtopia 组件支持$QTOPIA_PATH 。如，设置这个参数到 “/opt/Qtopia:/opt/Qtopia-ROM”，还需要设置 $PATH和$LD_LIBRARY_PATH。

8.3.9 MMS 客户端

Qtopia Phone Edition 版本包括在短消息客户端的 MMS 功能。

（1）WAP协议栈集成

Qtopia 不包括带有 MMS客户端的 WAP协议栈，但它提供了可集成任何 WAP 协议栈的接口。这个接口类必须从 MmsComms 类派生类，并实现其虚拟函数。例如，MmsCommsHttp 类提供了一个可

在 TCP/IP 协议上使用 HTTP 协议的样例，该类的实现代码见 $QPEDIR/src/applications/qtmail/mmscomms_http.cpp文件。

（2）选择可替代的 WAP 协议栈方法

通过修改 $QPEDIR/src/applications/qtmail/mmsclient.cpp 中的函数 MmsClient::setAccount 来选择 MmsComms 派生类，修改 $QPEDIR/src/applications/qtmail/qtmail.pri 中的宏定义 MMSCOMMS_HEADERS 和 MSCOMMS_SOURCES 分别指定 MmsComms 派生类的 .h 和 .cpp 文件。

（3）AMR 编码器

包括在 Qtopia 的AMR 编码器仅仅是一个用于 MMS 的参考编码器，建议安装一个目标平台优化的编码器。媒体记录器插件接口提供了一个集成编解器（codec）的方便方法，见 MediaRecorderEncoder 类。

8.3.10 GSM 模块集成

我们发现 GSM 规范的实现在不同的 GSM 模块之间是有变化的，因此很可能至少需要修改 $QPEDIR/src/libraries/qtopiaphone/{muxdevice.*, atchat.*} 文件来确保正在使用的模块是 GSM 规范的子集。

为了支持 GSM 模块，下面的指导可能被使用。

（1）检查 Qtopia 要求的 AT 命令并确定 GSM 模块支持或不支持的协议。

（2）决定适合 GSM 模块的可替换的合适 AT 命令，并修改需要的 phonelibrary 库。

（3）运行 Qtopia。

（4）检查报告出错命令的 error/message log 文本。

（5）使用 error log 修改 phonelibrary，以使用 modem 支持的 AT 命令参数，例如，AT+CMUX=1,0,5 vs AT+CMUX=0,0,5。

（6）重复步骤 3，直到 GSM 呼叫是可能的，SIM 详细信息能被读到等。

8.3.11 定制键盘

定制键盘说明如下。

（1）Qtopia 键的默认扫描码

几乎所有的 PDA 及 Phone 使用在常规键盘上没有的键，为了安装在一个设备时 Qtopia 能正常操作，需要重定义这些键。最简单的方法是让设备键盘驱动程序发送扫描码，然后 Qtopia 映射这些扫描码。

为了正常映射 US-101 键盘，表 8-4 列出了 Qt 键的说明及其在键盘模式和非键盘模式下的使用情况。

表 8-4 Qt 键的说明及其使用情况

扫描码	Qt 键值	描　述	键盘模式下	非键盘模式下
65	Key_Menu	应用程序菜单键	可选	不可用
67	Key_Back	Accept/Close 对话框键	要求	不可用
68	Key_Yes	Yes 键	可选	不可用
69	Key_No	No 键	可选	不可用
87	Key_Call	开始打电话键，接听进来电话	可选	不可用
88	Key_Hangup	结束打电话，拒绝来电	可选	不可用
116	Key_Select	选择菜单选项，checkbox 及按钮	必须	不可用

续表

扫描码	Qt 键值	描　　述	键盘模式下	非键盘模式下
120	F31	激活/关闭输入法	可选	可选
121	F32	与 PC 机设备同步软件同步设备	可选	可选
122	F34	电源/睡眠按钮	可选	必须
123	F35	开/关背景灯	可选	可选
124	Context1	第一个上下文按钮	可选	不可用
125	Context2	第二个上下文按钮	可选	不可用
126	Context3	第三个上下文按钮	可选	不可用
127	Context4	第四个上下文按钮	可选	不可用

还有两个其他键无默认映射，并且不应该使用。列出它们的原因仅为了兼容性，如表 8-5 所示。

表 8-5　无默认映射的 Qt 键说明

扫描码	Qt 键值	描　　述	键盘模式下	非键盘模式下
无	Key_F30	映射到需选择的窗口部件上的 space 键	不应用	可选
无	Key_F33	映射到需要 ok 的窗口部件的 enter 键	不应用	可选

根据要求的按钮，Qtopia 可被灵活设计。Qtopia 默认映射的按键键值及其说明如表 8-6 所示。

表 8-6　Qtopia 默认映射的按键键值及其说明

键　　值	功能说明
Key_Select	选择/激活/触发/编辑当前高亮的条目
Key_Context1	软键 1，在默认映射中映射到 Key_Menu 键
Key_Back	离开对话框，或在编译文本是为 Key_Backspace 键功能，并且不是实际的 Key_Backspace 键
Key_Call	回答或打电话
Key_Hangup	挂起电话或返回到主屏幕
Key_Up	向上巡航键，巡航到上一个条目或上一行
Key_Down	向下巡航键，巡航到下一个条目或下一行
Key_Left	向左巡航键，巡航到 Tab 窗口部件的前一页或左边一个字符
Key_Right	向右巡航键，巡航到 Tab 窗口部件的下一页或右边一个字符
Key_Flip	打开时，锁住屏幕，释放时，打开屏幕，回答或挂起电话
Key_Backspace	删除字符/条目

在没有 Key_Call 和 Key_Hangup，或者 Key_Left 和 Key_Right，或者 Key_Menu 时，Qtopia 可进行操作。

（2）重载 Qtopia 扫描码映射

当默认的 Qtopia 扫描码映射不能满足特定设备的需要时，可以重载 Qtopia 扫描码映射。这个重载在 src/libraries/qtopia/custom-linux-<device>-g++.cpp　中实现，在 custom-linux-<device>-g++.h 中加上下面的声明：

```
#define QPE_OVERRIDE_KEYMAP
struct KeyOverride;
extern const KeyOverride* qtopia_override_keys();
```

并在 custom-linux-<device>-g++.cpp　中加入扫描码重载：

```
typedef struct KeyOverride {
    ushort scan_code;
    QWSServer::KeyMap map;
};

static const KeyOverride deviceKeys[] = {
    //扫描码、扫描码的Qt键、扫描码字符、shift键激活时的字符及control键激活时字符
   { 0x79,   {   Qt::Key_F34,    0xffff, 0xffff, 0xffff  } }, // power
#if defined(QT_KEYPAD_MODE)
   { 0x7a,   {   Qt::Key_Call,      0xffff, 0xffff, 0xffff  } },
   { 0x7b,   {   Qt::Key_Context1,  0xffff, 0xffff, 0xffff  } },
   { 0x7c,   {   Qt::Key_Back,      0xffff, 0xffff, 0xffff  } },
   { 0x7d,   {   Qt::Key_Hangup,    0xffff, 0xffff, 0xffff  } },
   { 0x7e,   {   Qt::Key_Flip,      0xffff, 0xffff, 0xffff  } },
#else
   { 0x7a,   {   Qt::Key_F9,        0xffff, 0xffff, 0xffff  } },
   { 0x7b,   {   Qt::Key_F10,       0xffff, 0xffff, 0xffff  } },
   { 0x7c,   {   Qt::Key_F11,       0xffff, 0xffff, 0xffff  } },
   { 0x7d,   {   Qt::Key_F12,       0xffff, 0xffff, 0xffff  } },
   { 0x7e,   {   Qt::Key_F13,       0xffff, 0xffff, 0xffff  } },
#endif
   { 0,      {   0,                 0xffff, 0xffff, 0xffff  } }
};

const KeyOverride* qtopia_override_keys()
{
    return deviceKeys;
}
```

当屏幕锁被激活后，应用程序仅被允许使用诸如 Key_Back **和** Key_No **这样的键。**

（3）Qtopia 中的按键行为

有一些按键在到达一个应用程序之前总是被拦截的，如果有一个指针设备可用，应用程序会做一个键盘来得到这些键。总是被拦截的按键有 Key_F30、Key_F31、Key_F32 、Key_F33、Key_F34、Key_F35、Key_NumLock、Key_CapsLock、Key_Menu、Key_Call、Key_Hangup **和** Key_Flip。

有时被拦截的按键列出如下。

（1）Key_Escape**：离开或关闭激活对话框。**

（2）Key_F33 **接受激活的对话框。**

（3）Key_Back **：接受激活的对话框。**

（4）Key_Context1**：具有** Key_Menu **一样的行为。**

当屏幕锁激活时，应用程序仅被允许使用诸如 Key_Back **和** Key_No **这样的键。**

8.3.12 配置硬件按钮

一个设备的硬件按钮的配置定义在 etc/defaultbuttons-<platform-spec>.conf **文件中，这个文件被分成几个小节。**

（1）用户定义的按钮映射

按钮映射是将硬件按钮映射到一个应用程序快捷键，如日历按钮将打开“今天的计划”应用程序，或者一个按钮打开“事务卡”应用程序。按钮的描述包括在设备的 defaultbuttons.conf **文件或** defaultbuttons-<platform>.conf **文件中。**

一个硬件按钮的配置样例列出如下：

```
 [Button]
Count=2      #表示 有两组
[Button0]
```

```
Name=Calendar Button  # 按钮名，可通过加一项 Name[ja]=…来对名字提供翻译
Key=F9       #按钮映射的 Qt 键值
PressedActionService=Calendar #在
PressedActionMessage=raiseToday()
HeldActionService=Calendar
HeldActionMessage=raise()
[Button1]
Name=Contacts Button
Key=F10
# Qtopia 服务在 Qtopia Application Services 中描述
PressedActionService=Contacts  #    按下时的服务名
PressedActionMessage=raise()  #按下时的动作
HeldActionService=Contacts    #长时间按下时的服务名
HeldActionMessage=beamBusinessCard()  #    长时间按下时的动作
```

（2）手机应用程序 Launcher 菜单

手机应用程序 Launcher 菜单一般是 4 行 3 列的表格，按照位置顺序编上相应的键值 1~9、*及 0 和 #，即手机的 12 个键。

Launcher 菜单的配置样例如下：

```
[Menu]
Rows=3  #菜单的行数
Columns=3  #菜单的列数
Map=123456789  #菜单的位置号，从左到右，从上到下编号
Default=5    #默认号为 5
1=
2=Applications/datebook.desktop  #desktop 文件或目录
3=Games
4=
5=Applications/addressbook.desktop
6=Settings
7=
8=Applications
9=Documents
```

（3）手机按钮

一个手机的拨号及其他按钮在 defaultbuttons.conf 文件中的“ SystemButtons”、“TextButtons”、“PhoneTextButtons ”和“ LocaleTextButtons ”小节中分别进行配置，其中这个文件的 SystemButtons 小节的一个样例列出如下：

```
[SystemButtons]
Count=5
Key0=Menu
Key1=Select
Key2=Back
Key3=Call
Key4=Hangup
```

TextButtons 小节定义了与每个按钮相关的所有文本及动作，例如， 2 键与 2 相联系，如同 a、b、c 及其他这样的字符一样。

LocaleTextButtons 小节与 TextButtons section 一样，除了仅定义与当前语言相关的文本外，该小节中的文本按照配置文件定义的标准方式被翻译。

PhoneTextButtons 小节定义了所有的电话号码文本及其与每个按钮相联系的动作，例如， *键可能选择 *、+和 p（表示暂停）中的一个字符。

TextButtons、PhoneTextButtons 和 LocaleTextButtons 小节的格式相似，列出如下：

```
[TextButtons]
```

```
Buttons=0123456789*#
Tap0= ...
Hold0= ...
```

列在 Buttons 条目中每个按钮有 TapX 及 HoldX 两项定义，说明如下。

TapX 表示当键 X 被按下和释放时，键所执行的动作，其定义是下面各选项中的一种。

（1）'text：从文本中插入字符。

（2）"text：在显示所有选项时在文本中插入字符。

（3）space：插入一个空格。

（4）symbol：弹出字符选择器。

（5）modify：在可替换的字符中选择。

（6）mode 改变模式或语言。

（7）shift：切换大写/小写。

HoldX 表示当键 X 被按住时键所执行的动作，除了不支持 'text 项以外，与 Tap 一样。在第 1 个标点符号处插入字符被特殊处理，这个特殊处理的细节是内部定义的。

（4）手机 SoftKey 定义

SoftKey 定义小节的格式如下：

```
[SoftKeys]
Count=n
Key0=Name of Qt key code, e.g. Menu, Select, Context1
```

一个 SoftKey 定义小节的样例列出如下：

```
[SoftKeys]
Count=3
Key0-Context1
Key1=Select
Key2=Back
```

8.4 系统集成

系统集成是指安装 Qtopia 到一个新的设备中所需要做的工作，本节说明设备系统集成的各方面工作的次序。

8.4.1 创建设备特定交叉编译工具配置文件

特定设备将定义特定的交叉编译工具链，如一个新的基于 arm 的设备可能创建一个基于 qws/linux-arm-g++ 之上的 qws/linux-myarm-g++工具链配置。应将交叉编译工具链拷贝到特定设备的目录下，方法如下：

```
cp -r $QTEDIR/configs/linux-arm-g++-shared $QTEDIR/configs/linux-myarm-g++-shared
cp -r $QPEDIR/mkspecs/qws/linux-arm-g++ $QPEDIR/mkspecs/qws/linux-myarm-g++
```

8.4.2 定制电源、背景灯控制和键盘扫描码

有多个基本的 custom.cpp 文件，用户可在此基础上定制自己的 custom-*.cpp 文件，这些基本的 custom.cpp 文件基于 generic、innovator、ipaq 和 sharp 等设备。例如，用户在 sharp 设备的基础上定制

custom.cpp 文件，即首先拷贝这个文件，然后加入自己的特殊代码。拷贝方法如下：

```
cp $QPEDIR/src/libraries/qtopia/custom-linux-ipaq-g++.cpp
                        $QPEDIR/src/libraries/qtopia/custom-linux-myarm-g++.cpp
cp $QPEDIR/src/libraries/qtopia/custom-linux-ipaq-g++.h
                        $QPEDIR/src/libraries/qtopia/custom-linux-myarm-g++.h
```

定制键盘扫描码及按钮的细节参见 8.3.11 节。为了支持有刻度的触摸屏，必须在 custom.h 文件中定义宏 QPE_NEED_CALIBRATION。

8.4.3 在 Qt/Embedded 中创建键盘处理函数

根据内部的 client/server 协议，每个键的按下和释放作为一个 QWSKeyEvent 事件发出，QWSKeyEvent 事件包含下面几个域。

（1）unicode：Unicode 值。

（2）keycode：Qt 键编码值，定义在 qnamespace.h 中。

（3）modifier：由 Qt::ShiftButton、Qt::ControlButton 和 Qt::AltButton 组成的 bit 位的域。

（4）is_press ：如果键按下，为 TRUE ；释放则为 FALSE。

（5）is_auto_repeat：如果这个事件由自动重复引起，为 TRUE。

当 server 接收到一个键的事件，这个事件被送到每个客户进程，客户进程负责处理键事件并把事件送到正确的窗口。

（1）键盘驱动程序

键盘驱动从设备读取数据并把键的事件发送给服务器，当前被编译进库中，它定义在文件 src/kernel/qkeyboard_qws.cpp 中。几种键盘驱动的类列出如下。

（1）QWSTtyKeyboardHandler：从系统控制台（tty）输入。

（2）QWSVr41xxButtonsHandler ：从 Cassiopeia-style PDA 上的按钮输入。

（3）QWSVFbKeyboardHandler ：从虚拟帧缓冲窗口中的键输入。

键盘驱动从设备中读取键盘数据，找出哪个键被按下，然后调用静态函数 QWSServer::processKeyEvent()来处理按键事件。

目前，控制台键盘还能处理控制台切换（Ctrl+Alt+F1...F10 ）和控制台关闭（Ctrl+Alt+Backspace）。

为一个新设备增加键盘驱动，需要编写一个从 QWSKeyboardHandler 继承的子类，并在 qkeyboard_qws.cpp 文件的 QWSServer::newKeyboardHandler() 函数中进行实例化。

（2）按键事件过滤（输入法）

当 server 接收到从键盘驱动收到一个按键事件时，首先传递这个按键事件到一个事件过滤器。过滤器可被用做输入法，提供键盘上没有的字符的输入。

为了创建一个输入法，必须从 src/kernel/qwindowsystem_qws.h 中的 QWSServer::KeyboardFilter 函数得到的类派生一个子类，并实现派生类的虚拟函数 filter()。如果 filter() 返回 TRUE，事件将被停止。使用 QWSServer::sendKeyEvent()函数可产生新的按键事件，而不能使用 processKeyEvent()函数。因为已被用过，所以使用它将会导致无限的递归循环。

使用 QWSServer::setKeyboardFilter()安装一个键盘事件过滤器，每次只能安装一个过滤器。过滤操作必须在 server 进程中执行。

（3）触笔输入

在用触笔输入的设备中，键盘事件不必来自键盘驱动， server 进程可以在任何时候调用 QWSServer::sendKeyEvent()函数。典型的做法是弹出一个 widget 窗口部件，由用户通过指针设备选取特定的字符。这个输入法的 widget 不应该得到焦点，因为 server 然后把按键事件传回到用户操作的 widget，可通过在输入法的 widget 对应的 QWidget 类构造函数中设置 WStyle_Customize 和 WStyle_Tool

标识来实现。

为了增加一个新的驱动程序，必须修改文件 $QTEDIR/src/kernel/qkeyboard_qws.cpp，增加键盘驱动程序的方法见第 4 章 。

8.4.4 配置指针设备

在编译时用类似表 8-6 所示的宏来指定鼠标设备。

表 8-6 宏指定的鼠标设备

宏定义	指定的鼠标设备
QT_QWS_IPAQ 和 QT_QWS_IPAQ_RAW	/dev/h3600_tsraw
QT_QWS_IPAQ	/dev/h3600_ts
QT_QWS_SL5XXX	/dev/ts

（1）增加一个新的鼠标驱动

指针设备有触摸板（touchpanel）、跟踪球（trackball）或鼠标，Qt/Embedded 包括支持大量的指针设备，但运行中起作用的只有一个。

如果鼠标是/dev/psaux 或 dev/ttyS?串行线设备中支持的一种，Qt/Embedded 正常情况下自动检测鼠标类型。也可以设置环境变量 QWS_MOUSE_PROTO 来决定使用哪种鼠标，设置方法如下：

```
<protocol>:<device>
```

其中，<protocol>是 MouseMan、IntelliMouse 或 Microsoft 中的一种。<device>是鼠标设备，通常是/dev/mouse。如果这个参数未指定，默认为 Auto，即自动检测鼠标协议和鼠标设备。为了增加其他的鼠标协议，可以在 kernel/qwsmouse_qws.cpp 中写入 QAutoMouseSubHandler 或 QMouseHandler 的新子类。

（2）触摸板

Qt/Embedded 支持 NEC Vr41XX 和 iPAQ 触摸板，它们是 QCalibratedMouseHandler 的子类，而 QCalibratedMouseHandler 是在 kernel/qwsmouse_qws.cpp 中 QMouseHandler 的子类。

Qt/Embedded 在 $QTDIR/src/kernel/qwsmouse_qws.cpp 文件中的 QCustomTPanelHandlerPrivate 类中包括一个触摸板处理样例，样例从/dev/ts 中读出数据。数据包由 5 个字节组成，格式如下。

（1）Byte 0：给出状态信息，bit 6(0x40)为 1 表示鼠标键按下；为 0 表示释放。

（2）Bytes 1 和 2：给出 x 轴位置。

（3）Bytes 3 和 4：给出 y 轴位置。

为了激活这个指针驱动程序，建议在文件 qwsmouse_qws.cpp 中加入如下宏定义行：

```
#define QWS_CUSTOMTOUCHPANEL
```

有时候，触摸板设备不能精确匹配样例设备。例如，位于/dev/touchpanel 下的一个假定设备，它使用 6字节包，第 0个和第 1 个字节给出了状态和压力信息。特殊的是，第 1 个字节的第 5 位（0x20）告诉设备指针是上或下，第 2 个和第 3 个字节给出 X 轴位置，第 4 个和第 5 个字节给出 Y 轴位置。

压力信息对于基本的 Qt/Embedded 操作是不必要的，因此将忽略这个信息。下面列出了这个假定设备修改后的触摸板处理函数：

```
//修改了样例处理函数来处理假想的触摸板
QCustomTPanelHandlerPrivate::QCustomTPanelHandlerPrivate( MouseProtocol, QString )
{
    //打开 /dev/touchpanel 设备
```

```
    if ((mouseFD = open( "/dev/touchpanel", O_RDONLY)) < 0) {
        qWarning( "Cannot open /dev/touchpanel (%s)", strerror(errno));
        return;
    }

    QSocketNotifier *mouseNotifier;
mouseNotifier = new QSocketNotifier( mouseFD, QSocketNotifier::Read,
                                     this );
    connect(mouseNotifier, SIGNAL(activated(int)),this, SLOT(readMouseData()));
}

QCustomTPanelHandlerPrivate::~QCustomTPanelHandlerPrivate()
{
    if (mouseFD >= 0)
        close(mouseFD);
}

struct CustomTPdata {

  unsigned char status;
  unsigned short xpos;
  unsigned short ypos;

};

void QCustomTPanelHandlerPrivate::readMouseData()
{
    if(!qt_screen)   return;
    CustomTPdata data;
    //改变为 6 个字节大小
    unsigned char data2[6];
    int ret;
    //读取 6 个字节数据
    ret=read(mouseFD,data2,6);

    if(ret==6) { //取了 6 个字节
        //得到指针设备的状态、X 及 Y 轴坐标位置，忽略指针设备压力
        data.status=data2[1];   //设备状态
        data.xpos=(data2[2] << 8) | data2[3];
        data.ypos=(data2[4] << 8) | data2[5];
        QPoint q;
        q.setX(data.xpos);
        q.setY(data.ypos);
        mousePos=q;
        if(data.status & 0x20) { //*** Changed to 0x20 (bit 5)
          emit mouseChanged(mousePos,Qt::LeftButton);//按下/移动
        } else {
          emit mouseChanged(mousePos,0);  //释放
        }
    }
}
```

如果需要在运行时的不同的鼠标/触摸板设备之间切换，则必须修改函数 QWSServer::new MouseHandler()（在 qwsmouse_qws.cpp 中）来实例化新处理函数，并且必须增加鼠标协议号 MouseProtocol 和 mouseConfig[]表。

8.4.5 配置适合设备的 Qt/Embedded 库

不同设备根据设备平台及硬件的情况对 Qt/Embedded 库需要不同的配置，如运行于某些平台的 Qtopia 1.5 不需要线程支持。如果设备有较小的内存，则需要减小 Qt/Embedded 库，将其中不需要的构件删除。

为了满足用户对 Qt/Embedded 库的配置，Qtopia 在 $QPEDIR/src/qt/qconfig-qpe.h 文件中定义了 Qt/Embedded 库的默认配置。用户修改这个配置文件后，需要将其拷贝到 $QTEDIR/src/tools 目录，然后编译 Qt/Embedded 配置。

注意如果 Qt/Embedded 取消的组件过多，则有可能使在不同 PC 机交叉编译后的应用程序在设备上因缺少库函数而无法使用。

（1）qconfig-qpe.h 文件

下面是 qconfig-qpe.h 文件的部分代码，未列出的代码与其功能类似：

```
#ifndef QT_H
#endif // QT_H


#ifndef QT_DLL
#define QT_DLL // Internal
#endif
...
// 不需要 builtin codecs （Qtopia 使用插件增加私有的 codecs）
#ifndef QT_NO_CODECS
#define QT_NO_CODECS
#endif

// 不需要 builtin 字体工厂（Qtopia 使用插件来增加字体）
#define QT_NO_FREETYPE
#define QT_NO_BDF
...
```

（2）qfeatures.h 文件

config.h 文件简单地定义宏来取消某特征，一些特征依赖于其他的特征，这些依赖关系表达在 qfeatures.h 文件中。可用的选项如表 8-6~表 8-18 所示。

表 8-6 图像 QimageIO （含有装载和保存图像所需的参数）的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_IMAGEIO_BMP	微软的位图图像文件格式	
QT_NO_IMAGEIO_PPM	轻便的位图图像文件格式，即 Portable Pixmap	
QT_NO_IMAGEIO_XBM	X11 位图图像文件格式，即 X11 Bitmap	
QT_NO_IMAGEIO_XPM	X11 位图图像文件格式，即 X11 Pixmap	
QT_NO_IMAGEIO_PNG	轻便网络图片文件格式，即 Portable Network Graphics	

表 8-7 动画的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_ASYNC_IO	异步 I/O，-QAsyncIO	
QT_NO_ASYNC_IMAGE_IO	异步图像 I/O 及 GIF 图像支持等，-QImageDecoder	
QT_NO_MOVIE	动画支持，-QMovie	QT_NO_ASYNC_IO 和 QT_NO_ASYNC_IMAGE_IO

表 8-8 字体的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_TRUETYPE	TrueType 字体（TTF 和 TTC）文件格式，仅用于 Qt/Embedded	
QT_NO_BDF	位图分布格式字体文件格式，即 Bitmap Distribution Format（BDF），仅用于 Qt/Embedded	
QT_NO_FONTDATABASE	字体数据库	

表 8-9　字符国际化的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_I18N	在 Unicode 和 8 位字体编码之间转换	
QT_NO_UNICODETABLES	定义诸如所有 Unicode 字符大写与小写转换的大的表	

表 8-10　MIME 的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_MIME	在 internet 上网编码及标识各种类型数据的标准（如文本、图像及颜色等），-QMimeSource	
QT_NO_RICHTEXT	HTML 样式文本，-QStyleSheet,QLabel	QT_NO_MIME
QT_NO_DRAGANDDROP	在应用程序之间拖放的数据，-QDragObject	QT_NO_MIME
QT_NO_CLIPBOARD	在应用程序之间剪切粘贴的数据，-QClipboard	QT_NO_MIME

表 8-11　声音的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_SOUND	播放声音文件，-QSound	

表 8-12　Qt 的属性脚本选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_PROPERTIES	基于 Qt 应用程序的脚本	

表 8-13　Qt/Embedded　特定的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_QWS_CURSOR	在 Qt/Embedded 的光标，笔操作设备在正常情况下将不需要这个特征	
QT_NO_QWS_DEPTH_8GRAYSCALE	每像素点 8 位，256 级灰度	与 QWS_DEPTH_8 不相适应
QT_NO_QWS_DEPTH_8	每像素点 8 位，216 种颜色，40 种辅助颜色的方块	与 QWS_DEPTH_8GRAYSCALE 不相适应
QT_NO_QWS_DEPTH_15	每像素点 15 位，每种红、蓝和绿色分为 32 级	
QT_NO_QWS_DEPTH_16	每像素点 16 位，每种红和蓝色分为 32 级，绿色分为 64 级	
QT_NO_QWS_DEPTH_32	每像素点 32 位，每种红、蓝和绿色分为 256 级	
QT_NO_QWS_MACH64	Mach64 加速驱动程序（仅用于示范）	
QT_NO_QWS_VFB	运行于 X11 的虚拟帧缓冲	

表 8-14　网络的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_NETWORKPROTOCOL	使用包括在 QnetworkProtocol（网络协议基类，为网络协议提供通用 API）中的本地文件返回的抽象多协议数据	
T_NO_NETWORKPROTOCOL_FTP	返回的 FTP 协议数据	QT_NO_NETWORKPROTOCOL
QT_NO_NETWORKPROTOCOL_HTTP QT_NO_NETWORKPROTOCOL	返回的 HTTP 协议数据	

表 8-15 绘画的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_COLORNAMES	诸如红颜色名，被一些 QColor 构造函数和 HTML 文档使用，-QColor，QStyleSheet	
QT_NO_TRANSFORMATIONS	它使用了浮点运算，Qt 中的许多类用到了它。使用它旋转及放大才有可能；否则仅能坐标变换（即平移），-QWMatrix	
QT_NO_PSPRINTER	支持 Postscript 打印机	
QT_NO_PRINTER	支持打印机-Qprinter，在 Unix 上的 QT_NO_PSPRINTER	
QT_NO_PICTURE	保存 Qt 绘制命令到一个-QPicture 文件	

表 8-16 窗口部件的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_WIDGETS	除了 QWidget 外，取消所有的窗口部件	
QT_NO_TEXTVIEW	HTML 文档视图，-QTextView	QT_NO_WIDGETS, QT_NO_RICHTEXT
QT_NO_TEXTBROWSER	HTML 文档浏览器，-QTextBrowser	QT_NO_TEXTVIEW
QT_NO_ICONVIEW	标签化的图标，-QIconView	QT_NO_WIDGETS, QT_NO_DRAGANDDROP
QT_NO_LISTVIEW	信息的列表视图，-QListView	QT_NO_WIDGETS
QT_NO_CANVAS	对象画布，-QCanvas	QT_NO_WIDGETS
QT_NO_DIAL	值控制，-QDial	QT_NO_WIDGETS
QT_NO_WORKSPACE	多文档接口，-QWorkspace	QT_NO_WIDGETS
QT_NO_LCDNUMBER	LCD 样数字显示，-QLCDNumber	QT_NO_WIDGETS

表 8-17 GUI 模式的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_STYLE_WINDOWS	微软窗口模式，QWindowsStyle	QT_NO_WIDGETS
QT_NO_STYLE_MOTIF	OSF Motif 模式，QMotifStyle	QT_NO_WIDGETS
QT_NO_STYLE_CDE	开放组 CDE 模式，QCDEStyle	QT_NO_STYLE_MOTIF
QT_NO_STYLE_PLATINUM	MacOS X 模式，QPlatinumStyle	QT_NO_WIDGETS
QT_NO_STYLE_SGI	SGI 模式，QSGIStyle	QT_NO_STYLE_MOTIF

表 8-18 对话框的选项定义

宏定义	取消的特征说明	自动设置的特征说明
QT_NO_DIALOGS	取消所有的共同对话框，QWidget	QT_NO_WIDGETS
QT_NO_FILEDIALOG	文件选择对话框，-QFileDialog	QT_NO_DIALOGS, QT_NO_NETWORKPROTOCOL, QT_NO_LISTVIEW
QT_NO_FONTDIALOG	字体选择对话框，-QFontDialog	QT_NO_DIALOGS, QT_NO_FONTDATABASE
QT_NO_COLORDIALOG	颜色选择对话框，-QColorDialog	QT_NO_DIALOGS
QT_NO_PRINTDIALOG	打印选择对话框，-QPrintDialog	QT_NO_DIALOGS, QT_NO_LISTVIEW
QT_NO_INPUTDIALOG	文本输入对话框，-QInputDialog	QT_NO_DIALOGS
QT_NO_MESSAGEBOX	消息/弹出对话框，-QMessageBox	QT_NO_DIALOGS
QT_NO_PROGRESSDIALOG	长计算进度条对话框，-QProgressDialog	QT_NO_DIALOGS
QT_NO_TABDIALOG	Tab 页对话框，-QTabDialog	QT_NO_DIALOGS
QT_NO_WIZARD	多步向导对话框，-QWizard	QT_NO_DIALOGS

在 qfeatures.h 中部分选项的定义列出如下：

```
//没有 Unicode 文本转换
#if !defined(QT_NO_CODECS) && (defined(QT_NO_TEXTCODEC))
#define QT_NO_CODECS
#endif

// QCop IPC
#if !defined(QT_NO_COP) && (defined(QT_NO_DATASTREAM))
#define QT_NO_COP
#endif

// QFontDatabase
#if !defined(QT_NO_FONTDATABASE) && (defined(QT_NO_STRINGLIST))
#define QT_NO_FONTDATABASE
#endif

// Image 格式
#if !defined(QT_NO_IMAGEIO) && (defined(QT_NO_REGEXP))
#define QT_NO_IMAGEIO
#endif
```

（3）Qt/Embedded 配置选项

Qt/Embedded运行 configure 脚本时，选择适合设备的平台命令选项，即 -xplatform linux-myarm-g++。

8.4.6 配置适合设备的 Qtopia

为了使 Qtopia 正常运行，Linux 内核必须提供共享内存、mmap 映射和 socket，并且必须提供帧缓冲驱动程序、OSS 声音支持或与 OSS 相适应的 ALSA 声音支持。另外还需要库 Video4Linux、zlib、libuuid （aka luuid）、libjpeg、xorg 或 x11.

应该在 custom-<platform-spec>.cpp 中定义硬件特定的软件组件，如输入法、电源管理及可移去的卡管理等,在 src/tools/qconfig.h 中定义 Qtopia 功能的子集。

8.4.7 为目 标设备创建和安装 Qtopia image 文件

当为目标设备配置 Qtopia 时，$QPEDIR/image 是 Qtopia 编译安装后的映像文件所在的默认目录，即 image 目录下所在的文件及目录结构为 Qtopia 在目标设备上运行的文件及目录结构。可以使用 NFS 文件系统或者将 image 目录下文件及目录打包，然后经设备特定工具软件烧录到目标设备中。

8.4.8 在设备上配置适合于 Qtopia 的环境变量

在设备上如下设置 Qtopia 运行的环境变量：

```
export QPEDIR=/opt/Qtopia
export QTDIR=$QPEDIR
export QTEDIR=$QTDIR
export PATH=$QPEDIR/bin:$PATH
export LD_LIBRARY_PATH=$QPEDIR/lib:$LD_LIBRARY_PATH
```

8.4.9 安装另外的字体

使用工具 makeqpf 来创建字体文件，这个工具的使用方法如下：

```
makeqpf [-A] [-f spec-file] [font ...]
```

makeeqpf 命令说明如下。

（1）-A ：提交处理并存储在 fontdir 目录下所有的字体。

（2）-f：特定的字体文件名：字体名字符范围，如 smoothtimes 0-ff,20a0-20af。

（3）font：提交并存储的字体。

8.4.10 集成 Java

当集成一个 Java VM 虚拟机到Qtopia 时，安装的 Java 应用程序可以与 Qtopia 应用程序一起操作，终端用户感觉不到 Java VM 的存在。Java 应用程序配置方法说明如下。

（1）使用 .desktop 文件

如同 Qtopia 应用程序一样，Java 应用程序应在 $QPEDIR/apps/Applications 或 $QPEDIR/apps/Games 目录（依赖于 Java 应用程序是应用程序还是游戏）中被 .desktop 文件引用。

（2）使用惟一的执行程序

为了保证 Qtopia 的执行模式工作，在 .desktop 文件中的 Exec=行应该定义一个惟一执行程序名，当然它可以是链接到一个实际的执行程序名的符号链接。对于在应用程序与 Launcher 之间的 QCOP 通信来说，这是基本的。特别是，使用“ Exec=jvm appname” 将不工作。

（3）使用 QPEApplication 类

必须使用 QPEApplication 类，而不是 QApplication 类 否则所有提供给 Qtopia 的框架将不起作用。特别是，QPEApplication 与 Launcher 通过 QCOP 通信来处理诸如任务栏图标、升起、退出、打开文档及改变风格和字体这样的功能。

（4）使用 QPEApplication::exec()

使用 QPEApplication::exec()保证诸如初始化 QCOP 信道等操作。如果必须使用自己的事件循环，而不是 exec()，则至少必须调用 QPEApplication::setMainWidget()。如果这样也不可能，则必须增加其他功能函数到 QPEApplication 中来支持自己的事件循环。

（5）遵循 Qtopia 协定

被 JVM 使用的窗口部件工具或绑定应该遵循 Qtopia 的协定，例如应用使用 QPEApplication:: execDialog()，而不是 QDialog::exec()。

8.5 Qtopia 编译系统

Qtopia 通过 .pri 和 .pro 文件对编译系统进行分层管理，这两个文件都是 qmake 使用的项目文件。二者格式一样，.pri 文件被 .pri 或 .pro 文件包括(使用 include 语句)，.pro 文件直接被用来产生 Makefile 文件。

Qtopia 项目文件的层次及 configure 脚本生成的 Makefile 层次如图 8-2 所示。

configure 脚本通过用户选择的编译选项生成 config.pri 项目文件，在编译的过程中会用这些编译选项。.qmake.cache 用来重定向到 config.pri 文件，configure 脚本使用 qmake 命令将各个目录下的 .pro 文件生成 Makefile.target 。顶层的 Makefile.src-component 包括相互无依赖关系的应用程序或插件的项目，Makefile.target 是应用程序所依赖的库的项目，Makefile 是用户在 configure 脚本中手动配置的编译入口选项。

在使用 make 命令进行编译时，编译器先从 Makefile 找到编译入口。然后使用 Makefile.target 编译被依赖的库，同时 Makefile.target 文件中使用 make 来调用 Makefile.src-component 来编译无依赖关系的应用程序及插件。

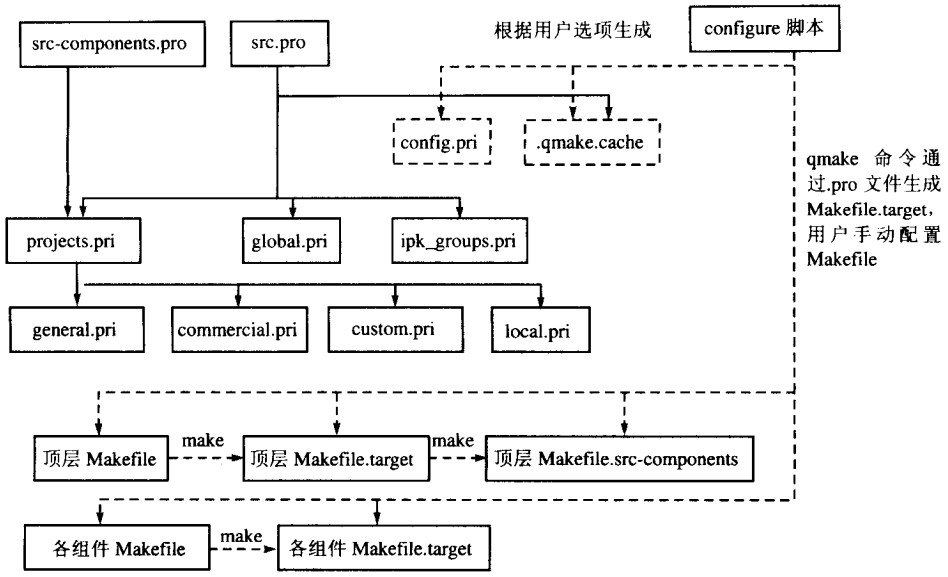


图 8-2 Qtopia 项目文件的层次及 configure 脚本生成的 Makefile 层次

8.5.1 内部的编译系统项目文件说明

Qtopia 的顶层项目文件为 src.pro 文件，其中包括多个 .pri 项目文件，这些 .pri 文件用来定义哪些源代码文件需要编译。并且提供了分层管理，方便编译时的文件选择。与生成 Qtopia 的顶层 Makefile 相关的项目文件说明如下。

（1）$QPEDIR/src/config.pri： 由 configure 脚本产生，该脚本将用户选定的选项转换为参数变量。config.pri 在其他每个项目文件处理之前被处理，其值可被重写。

（2）src/global.pri: 在其他每个项目文件处理之后被处理，其中保存可用的设置，如如何编译 ipkg 及连接通用库等。

（3）src/ipk_groups.pri： 在大多情况下，一个项目为一个 ipkg。有时候多于一个项目在一个 ipkg 中，这个文件定义了如何编译这样的一个包。

（4）projects.pri 管理来自 general.pri、commercial.pri、custom.pri 和 local.pri 的一个项目链表。

（5）src.pro 和 src-components.pro ：指定编译的项目。src.pro 文件管理需要以特定次序编译的项目，而 src-components.pro 文件管理可以任何次序被编译的项目。来自 src.pro 的项目在来自 src-components.pro 项目之前被编译。

src.pro 是用来产生顶层的 Makefile，并且知道如何编译在控件文件中的 ipkg，因为它们没有.pro 文件。

projects.pri 文件中包括的.pri 文件说明如下。

（1）general.pri ：链接在所有 Qtopia 源代码包中的构件，包括所有 GPL 和商业版下发布的 Qtopia 核心源代码。

（2）commercial.pri ：链接商业版特有的 Qtopia 源代码包。

（3）custom.pri ：链接用户配置部分的源代码包。

在上面几个.pri 项目文件中，通过定义几个项目链表变量，将各种源代码分类链接在一起。项目链表按次序处理，这样所有的库在应用程序之前编译。项目链表说明如表 8-19所示。

表 8-19 项目链表说明

项目链表	说 明
LIBRARY_PROJECTS	应用程序和插件用的库文件源代码链表
APP_PROJECTS	应用程序源代码链表
PLUGIN_PROJECTS	插件和 java 程序的源代码链表
THEME_PROJECTS	主题相关的链表

（1）projects.pri 文件

projects.pri 文件将不同类型的源代码分别使用不同的 .pri 文件来描述，projects.pri 文件管理这几个 .pri 文件，列出如下：

```
!contains( processed_pri, projects.pri ) {
    processed_pri += projects.pri

    include($${QTOPIA_DEPOT_PATH}/src/general.pri)
    include($${QTOPIA_DEPOT_PATH}/src/commercial.pri)
    include($${QTOPIA_DEPOT_PATH}/src/custom.pri)
    include($$(QPEDIR)/src/local.pri)

    LIBRARY_PROJECTS=$$CORE_LIBRARY_PROJECTS $$LIBRARY_PROJECTS

    # 如果 LIBRARY_PROJECTS 中包括 libraries/handwriting ，则因为重复必须删除一个
    !contains(LIBRARY_PROJECTS,libraries/handwriting) {
     PLUGIN_PROJECTS-=plugins/inputmethods/handwriting\
          plugins/inputmethods/fshandwriting
        APP_PROJECTS -= settings/handwriting
}
    ...
}
```

（2）config.pri 文件

config.pri 文件在运行 configure 脚本的过程中产生，它用变量说明用户给出的选项，该文件的一部分说明如下：

```
!contains( PROCESSED_PRI, config.pri ) {#确保 include 进一次
    PROCESSED_PRI += config.pri

    QTOPIA_DEPOT_PATH=/home/test/qt_soft/qtopia-free-2.1.1  #源代码所在目录
    QTOPIA_ICON_SIZES=14 16 22
    QTOPIA_DISP_SIZES= P
    QTOPIA_DISP_WIDTH=
    QTOPIA_DISP_HEIGHT=
    QTOPIA_DISP_ROTS=0 90 180 270
    CONFIG += warn_on #打印警告信息
    CONFIG+=debug   #编译成调试版本
    ...
    QMAKE_POST_INCLUDE_FILES=$${QTOPIA_DEPOT_PATH}/src/global.pri
    LANGUAGES=en_US
    ...
    QPE_VERSION=2.1.1

    CONFIG+=buildQtopia #编译为 Qtopia 版本
    CONFIG+=defaultbuttons
    QTOPIA_ARCHITECTURE=x86  #是能 x86 下运行的版本
    QTOPIA_ARCH=x86

    QCONFIGDEFINE=QCONFIG=\"qconfig-qpe.h\"
    ...
```

```
    CONFIG+=qt3
    include($${QTOPIA_DEPOT_PATH}/src/projects.pri)
}
```

（3）ipk_groups.pri 文件

ipk_groups.pri文件定义了安装插件相关的变量名， 其中一个 calc 包的定义列出如下：

```
#calc 包
QTOPIA_CALC_PACKAGE=libraries/qtopiacalc\
applications/calculator\
plugins/calculator/simple
contains(QTOPIA_CALC_PACKAGE,$${QTOPIA_ID}) {
    PACKAGE_NAME=qpe-calculator # 插件名
    IPK_GROUP_PROJECTS=$${QTOPIA_CALC_PACKAGE}
    #插件的描述
    PACKAGE_DESCRIPTION=A pluggable calculator for the Qtopia environment that includes a simple
interface.
}
...
```

（4）general.pri 文件

general.pri文件定义了需要加入的普通项目，即需要加入的核心应用程序及其所需要的库和插件该文件的一部分列出如下：

```
SERVER=server
#最基本的库，必须加入
CORE_LIBRARY_PROJECTS += qt libraries/qtopia
CORE_LIBRARY_PROJECTS+=libraries/qtopia1
...
!QTOPIA_CORE {#核心库
    LIBRARY_PROJECTS+=\
     libraries/qtopiapim\
     libraries/mediaplayer\
     libraries/qtopiacalc
    !QTOPIA_PHONE:THEME_PROJECTS+=mediaplayer/techno
    LIBRARY_PROJECTS+=libraries/qtopiapim1
}
...
!QTOPIA CORE {#核心应用程序
    APP_PROJECTS+=\
     applications/addressbook \
     applications/calculator \
     applications/camera \
     applications/clock \
     ...
}
...
PLUGIN_PROJECTS+=\
    plugins/decorations/flat\
    plugins/fontfactories/freetype\
    plugins/imagecodecs/notepad\
    ...
QTOPIA_PDA {#PDA  需要加入的应用程序及插件
    APP_PROJECTS+=\
     applications/today \
     games/mindbreaker\
     settings/appservices \
     settings/buttoneditor \
     ...
    PLUGIN_PROJECTS+=\
     plugins/applets/batteryapplet\
     plugins/applets/brightness\
```

```
        plugins/applets/clipboardapplet\
        plugins/applets/clockapplet\
        ...
    }
...
```

（5）global.pri 文件

global.pri 项目文件在其他每个项目文件后执行，其中保存各种全局设置，如如何编译 ipkg 和连接通用库及库之间的依赖关系等。该项目文件的一部分列出如下：

```
!contains( PROCESSED_PRI, global.pri ) {#确保 include 进一次
    PROCESSED_PRI += global.pri

    include($${QTOPIA_DEPOT_PATH}/src/projects.pri)
    #将所有的项目放在一个链表中
    ALL_PROJECTS=$${LIBRARY_PROJECTS} $${APP_PROJECTS} $${PLUGIN_PROJECTS} $${SERVER} $${THEME_
PROJECTS}
    #各种通用库的连接方法，这里的依赖关系转换为 Makefile.target 中库的依赖关系
    !contains(TEMPLATE,subdirs) {
       qtopiaapp:CONFIG+=qtopia
       qtopia:CONFIG+=qtopialib qtopiainc

     # 加入基本库的连接到 LIBS 中
     qtopiadesktop {
        CONFIG += desktop
        !contains(TARGET,qdbase) {
            LIBS        = -lqd-qdbase $$LIBS
            win32:DEFINES += QDBASE_DLL
        }
        qtopialib:LIBS+=-lqd-qpe
        ...
    }
     ...
     #设置路径
     CONFIG+=link_prl create_prl depend_prl
     qtest:INCLUDEPATH+=$$(QTESTDIR)/qtestlib
     ...
     # Ipk groups
     include ($${QTOPIA_DEPOT_PATH}/src/ipk_groups.pri)
     ...
     desktop {
        requires(!isEmpty(DQTDIR))
        CONFIG+=thread
        INCLUDEPATH=$${DQTDIR}/include $${INCLUDEPATH}
        DEPENDPATH=$${DQTDIR}/include $${INCLUDEPATH}

        win32:QMAKE_MOC=$${DQTDIR}\bin\moc.exe
        else:QMAKE_MOC=$${DQTDIR}/bin/moc
        win32:QMAKE_UIC=$${DQTDIR}\bin\uic.exe
        else:QMAKE_UIC=$${DQTDIR}/bin/uic
        QMAKE_INCDIR_QT=$${DQTDIR}/include
        QMAKE_LIBDIR_QT=$${DQTDIR}/lib
     }
    ...
     # Packaging
     qtopiadesktop {
        packages.commands=$${COMMAND_HEADER}echo "Not building packages for QtopiaDesktop"
     }
     ...
    sdk.depends+=install
    devsdk.depends+=sdk
    QMAKE_EXTRA_UNIX_TARGETS+=packages sdk devsdk
    !isEmpty(DQTDIR):QMAKE_EXTRA_UNIX_TARGETS+=lupdate
}
```

8.5.2 项目文件生成 Makefile

所有的 qmake 编译命令定义 qws/*（如 QMAKESPEC=qws/linux-generic-g++）在 $QPEDIR/src/config.pri 文件中有一个明确的指定，来自 Qtopia 源代码树的项目接受这个规则。但还需要处理不遵守这个规则的外部的项目，并且 Qtopia 桌面项目不使用 qws/* 这样的 QMAKESPEC 值。为了处理这种情况，configure 运行时将编译器选择配置写入到 $QPEDIR/src/.qmake.cache 文件中。文件 .qmake.cache 设置一个标识并用 include 语句包含 $QPEDIR/src/config.pri 文件，qmake 通过访问 .qmake.cache 文件来访问项目文件 config.pri。

config.pri 提供了 qmake 执行的命令参数选项（由用户指定），编译器在处理主项目文件 src.pro 后处理 global.pri 文件。重要的 .pri 文件有保护语句，确保其仅被执行一次。

通过包含 .pri 文件可以简化 Qtopia 和 Qtopia Desktop 的编译，一个典型的项目可能有 3 个项目文件，即一个 .pri 文件和两个 .pro 文件，下面以 src/libraries/qtopiapim 为例子说明。

qtopiapim.pri 列出所有 Qtopia 应用程序和 Qtopia Desktop 之间共同的文件，qtopiapim.pro 包括 qtopiapim.pri，并加入 Qtopia 特定的配置。src/qtopiadesktop/libraries/qtopiapim.pro 包括 qtopiapim.pri，并加入了 Qtopia Desktop 特定的配置。

configure 脚本执行后产生 Makefile 文件，当输入 make 命令读取 Makefile 文件时，Makefile 文件决定编译需要做的工作。由于 qmake 执行后的输出与 Qtopia 的需要有一些差别，因此 qmake 产生的文件命名为“Makefile.target”。

尝试手动再次创建 Makefile 是不安全的，因为可能重写 configure 脚本创建的特定的 Makefile。重运行 configure 脚本是创建 Makefile 和 Makefile.target 的惟一方法，然而为了允许更简单的开发，可以运行 make qmake 来再产生 Makefile.target 文件，使用 make regenerate 产生 Makefile 文件。

特定的 Makefile 指定特定的安装目录，参数 PREFIX 和 DPREFIX 分别指定 Qtopia 和 Qtopia Desktop 的安装位置。configure 把安装位置的默认值放在特殊的 Makefile 中，但可通过命令行重写安装位置，如 make PREFIX=/opt/Qtopia DPREFIX=/opt/QtopiaDesktop。

为了允许最大的弹性，特殊的 Makefile 使用定义在 Makefile.target 中的所有目标。如果 Makefile.target 被重建，目标将不再与 Makefile 中目标相匹配，运行 make regenerate 将矫正这个错误。

特定的 Makefile 将 PREFIX（或 DPREFIX）转换为 INSTALL_ROOT，因为 INSTALL_ROOT 是 qmake 为安装 prefix 使用的变量。如果没有一个特殊的 Makefile（如因为在 $QPEDIR 以外的目录手动运行 qmake），则必须设置 INSTALL_ROOT 到安装的位置（如 INSTALL_ROOT=/opt/Qtopia）。更好的方法是传递这个目录到 configure 中，如 configure -build /path/to/my/extra/stuff。

8.5.3 src-components.pro 和 src.pro 文件

src-components.pro 和 src.pro 文件说明如下。

（1）src-components.pro 文件

src-components.pro 文件包括与编译次序无关的项目，一般是一些独立的应用程序和插件，该文件列出如下：

```
TEMPLATE=subdirs

include($${QTOPIA_DEPOT_PATH}/src/projects.pri)

# 与编译次序无关的项目
SUBDIRS+=$${APP_PROJECTS} $${PLUGIN_PROJECTS} $${SERVER}
!isEmpty(THEME_PROJECTS):SUBDIRS+=../etc/themes
!isEmpty(EXTRA_SUBDIRS):SUBDIRS+=$${EXTRA_SUBDIRS}
```

（2）src.pro 文件

src.pro 文件用来产生顶层的 Makefile，它使用 general.pri 文件定义需要编译的核心应用程序及其对应的库和插件，src.pro 文件定义作为 SDK 所需要的应用程序、库和字体及其安装路径等。

src.pro 文件分析如下：

```
#这是个特殊的模板，它可以创建一个能够进入特定目录并且为一个项目文件生成 makefile，还调用 make
TEMPLATE=subdirs

#如果 buildQtopia 条件为真，编译 Qtopia
buildQtopia {
    #$$表示得到变量的值
    include($${QTOPIA_DEPOT_PATH}/src/projects.pri)  #将 .pri 文件包含其中
    CONFIG+=ordered
    #在使用 subdir 模板时，SUBDIRS存放有所有查找项目文件的子目录
    SUBDIRS+=$${LIBRARY_PROJECTS} src components.pro
}

buildQtopiadesktop:SUBDIRS+=qtopiadesktop
#定义软件包
nonpropackages.commands=$${COMMAND_HEADER}\
mkdir -p $$(QPEDIR)/ipkg;\  #如果目录不存在，创建目录
cd $$(QPEDIR)/ipkg;\
for i in \ #循环，i 为找到的文件名
#在 qt 目录下按照文件名查找*.control 文件
`find $${QTOPIA_DEPOT_PATH}/src/qt -name '*.control'` \
`find $${QTOPIA_DEPOT_PATH}/services -name '*.control'` \
`find $${QTOPIA_DEPOT_PATH}/sounds -name '*.control'` \
`find $${QTOPIA_DEPOT_PATH}/misc -name '*.control'`; do\
    echo Packaging \$$i;\
    $${QTOPIA_DEPOT_PATH}/bin/mkipks \$$i;\  #打包
done;

nonpropackages.CONFIG+=phony no_path
buildQtopia {
    QMAKE_EXTRA_UNIX_TARGETS += nonpropackages
    packages.depends = nonpropackages  #依赖关系
}

nonpropackagesinstall.commands=$${COMMAND_HEAI
for i in \ #按照文件名查找*.control文件
    `find $${QTOPIA_DEPOT_PATH}/services -name '*.control'` \
    `find $${QTOPIA_DEPOT_PATH}/sounds -name '*.control'` \
    `find $${QTOPIA_DEPOT_PATH}/misc -name '*.control'` \
;do \
    echo Installing files listed in \$$i;\
    #从 \$$i 文件中找到头部为 Files:的行(^表示行头可以有空格),并通过 sed 命令将 Files: 替换为空格，然后再传给 FILELIST
    FILELIST=`grep "^Files:" \$$i | sed -e 's/^Files://'`;\
    [ -z "\$$FILELIST" ] || for j in \$$FILELIST; do \
     dir=`dirname \$$j`;\
     mkdir -p "$(INSTALL_ROOT)/\$$dir";\    //如果目录不存在、创建目录
  #INSTALL_ROOT 为 opt/Qtopi，安装拷贝\$$j 到\$$dir 下，并设置属性
     eval install -c "$$(QPEDIR)/\$$j" "$(INSTALL_ROOT)/\$$dir";\
    done;\
done;\

nonpropackagesinstall.CONFIG+=no_path
buildQtopia:INSTALLS+=nonpropackagesinstall

# 安装拷贝字体文件 fonts为字符串链表数组，保存字体的名字、大小及安装路径等的字符串链表
contains(QTOPIA_DISP_ROTS,0):fonts.rotations={50,50i,75,75i}
contains(QTOPIA_DISP_ROTS,90):fonts.rotations+=*_t5
contains(QTOPIA_DISP_ROTS,180):fonts.rotations+=*_t15
contains(QTOPIA_DISP_ROTS,270):fonts.rotations+=*_t10
```

```
QTOPIA_PHONE {#如果 QTOPIA_PHONE 条件为真
    # 仅 Helvetica 和固定字体，字体大小为 8.0 pt~18.0 pt
    fonts.sizes={80,100,120,140,180}
}
!QTOPIA_PHONE {
    # 更多的字体，所有尺寸
    fonts.files+=micro
    fonts.files+=smallsmooth
    fonts.files+=smoothtimes
    fonts.sizes=*
}
buildQtopia {
    fonts.files+=helvetica
    fonts.files+=fixed
    fonts.path=/lib/fonts  #安装的路径为 /lib/fonts 目录
    fonts.commands=$${COMMAND_HEADER}\
     [ -z "$$fonts.rotations" ] || for rt in $${fonts.rotations}; do\
        [ -z "$$fonts.files" ] || for f in $${fonts.files}; do\
       #组合成字体文件名，得到诸如 helvetica_120_75i_t15.qpf 这样的文件名
       #$QTEDIR/lib/fonts  下的指定文件名字体文件拷贝到 $QTOPIA_PATH/lib/fonts 目录下
          for file in $${QTEDIR}/lib/fonts/\$${f}_$${fonts.sizes}_\$${rt    }.qpf; do\
              if [ -f \$$file ]; then\  #如果是文件
            #安装拷贝到 $ QTOPIA_PATH/lib/fonts  目录下
                install -c \$$file $(INSTALL_ROOT)$${fonts.path};\
              fi;\
          done;\
        done;\
     done
    INSTALLS+=fonts
}

#定义编译 SDK 的项目文件及应用程序，SDK 是运行 Qtopia 所需要的类库及工具应用程序等
!win32:!mac { # 如果不是 win32 平台或 mac 平台
    #设置执行 make install 时 qmake 的 INSTALL 变量，指定安装的文件源及安装路径
    #定义 qmake 需要安装拷贝的文件
    sdk_qmake.files=\    # 在源代码中需要安装拷贝的文件
     $$(QPEDIR)/bin/qmake\
     $$(QPEDIR)/bin/nct_lupdate\
     $$(QPEDIR)/bin/installpic\
     $$(QPEDIR)/bin/mkipks
    sdk_qmake.path=/bin  # 安装拷贝的目的路径
    sdk_qmake.CONFIG=no_default_install

    #定义不同平台编译器的选择路径
    sdk_mkspecs.files=$${QTOPIA_DEPOT_PATH}/mkspecs
    sdk_mkspecs.path=/
    sdk_mkspecs.CONFIG=no_default_install
    #定义编译时用到的 include 文件及路径
    sdk_qt_includes.files=$${QTEDIR}/include/*
    sdk_qt_includes.path=/include
    sdk_qt_includes.CONFIG=no_default_install

    sdk_pri.files=\    #定义 pri 项目文件的路径
     $${QTOPIA_DEPOT_PATH}/src/config.pri\
     $${QTOPIA_DEPOT_PATH}/src/global.pri\
     $${QTOPIA_DEPOT_PATH}/src/projects.pri\
     $${QTOPIA_DEPOT_PATH}/src/general.pri\
     $${QTOPIA_DEPOT_PATH}/src/ipk_groups.pri\
     $$(QPEDIR)/src/.qmake.cache  #这个文件中有 include ($$QPEDIR)/src/config.pri 语句
    !free_package:sdk_pri.files+=$${QTOPIA_DEPOT_PATH}/src/commercial.pri
    sdk_pri.path=/src
    sdk_pri.CONFIG=no_default_install

    #定义样例应用程序的路径
    sdk_examples.files=$${QTOPIA_DEPOT_PATH}/examples
```

```
        sdk_examples.path=/
        sdk_examples.CONFIG=no_default_install

        #将定义好的文件路径字符串数组变量加入到 INSTALL 中
        INSTALLS+=sdk_qmake sdk_mkspecs sdk_qt_includes sdk_pri sdk_examples
        sdk.depends+=install_sdk_qmake  install_sdk_mkspecs  install_sdk_qt_includes  install_sdk_pri
install_sdk_examples

        #定义安装 qtutils (Qt 应用程序) 字符串链表数组
        !isEmpty(QTEDIR) {
            sdk_qtutils.files=\  #定义 Qt/Embedde拷贝安装的应用程序文件名
                $${QTEDIR}/bin/moc\
                $${QTEDIR}/bin/uic\
                $${QTEDIR}/bin/designer\
                $${QTEDIR}/bin/qvfb

            !exists($${QTEDIR}/bin/qvfb):sdk_qtutils.files+=$${QTEDIR}/tools/qvfb/qvfb
            QTOPIA_PHONE:sdk_qtutils.files+=\  #定义手机版本的 qvfb运行设备仿真图形文件
                $${QTEDIR}/tools/qvfb/SmartPhone.skin\
                $${QTEDIR}/tools/qvfb/TouchscreenPhone.skin
            !QTOPIA_PHONE:sdk_qtutils.files+=\  #  定义 PDA 版本的 qvfb运行设备仿真图形文件
                $${QTEDIR}/tools/qvfb/QtopiaPDA.skin
            sdk_qtutils.path=/bin  #定义安装路径
            sdk_qtutils.CONFIG=no_default_install

           #将定义的文件路径字符串数组变量加入到 INSTALL 中
            INSTALLS+=sdk_qtutils
            sdk.depends+=install_sdk_qtutils
        }
        #定义国际化相关的应用程序文件及安装路径
        !isEmpty(DQTDIR) { #DQTDIR    不为空
            sdk_qt3utils.files=\  #定义国际化相关的应用程序文件名到变量中
                $${DQTDIR}/bin/lrelease\
                $${DQTDIR}/bin/lupdate\
                $${DQTDIR}/bin/linguist
            sdk_qt3utils.path=/bin  # 定义安装路径为 /bin ( 即$QTOPIA_PATH/bin)
            sdk_qt3utils.CONFIG=no_default_install
           #将定义的文件路径字符串数组变量加入到 INSTALL 中
            INSTALLS+=sdk_qt3utils
            sdk.depends+=install_sdk_qt3utils  #     依赖关系
        }
```

8.5.4 configure 脚本生成 Makefile

应用程序的编译及安装需求在 .pro 文件中都已指定，通过 qmake 命令生成 Makefile 后，即可使用 make命令进行编译，使用 make install 将生成的应用程序及相应的文件拷贝到指定的安装目录下。

configure 脚本文件用 perl 语言编写，它在对 configure参数及.pro 文件等进行各种配置检查后，调用 qmake 工具来生成 Makefile 文件。

在 configure文件中调用 qmake 相关的代码如下：

```
my $makefile = "Makefile.target";
$QMAKE_QTOPIA_EXTRA_ARGS .= " -o $makefile";
$QMAKE_QTOPIA_EXTRA_ARGS .= " -spec $depotpath/mkspecs/qws/$opt_xplatform"
my $qmake_cmd = "$outpath/bin/qmake $QMAKE_QTOPIA_EXTRA_ARGS $aProFile";
system($qmake_cmd)
```

展开上面这几行代码，得到类似下面的代码：

```
qmake -o Makefile.target -spec xscale src.pro
```

为了允许最大的弹性，Makefile使用了在 Makefile.target 中定义的所有目标。在 configure 文件中，将对 Makefile.target 文件中的各行进行分析处理后的内容写入 Makefile 文件，完成后由 Makefile.target 重定向生成 Makefile 的过程。

configure脚本运行的过程如下。

（1）设置各种编译中用到的临时变量。

（2）设置 configure 默认参数，并检查参数的有效性。

（3）建立安装目录体系，并将需要的非编译生成文件拷贝到指定的目录下。

（4）使用 find 查找命令或其他方法得到各个.pro文件名后放入一个数组中。

（5）使用 qmake 命令从.pro文件生成临时 Makefile.target，它是真正的 makefile文件。

（6）用户书写 Makefile 文件，使 Makefile 定向到 Makefile.target ，这样 make 命令通过 Makefile.target 来编译生成目标文件。

在 Makefile 中重定向 Makefile.target 的语句列出如下：

```
$(MAKE) -f Makefile.target $@ INSTALL_ROOT=$(PREFIX);
```

configure 脚本文件对于参数选项的分析很多，如下列出由 .pro 生成 Makefile 的部分代码：

```
#!/usr/bin/perl -w
#
my $cwd = getcwd(); //得到当前目录
my $outpath = $ENV{'QPEDIR'};
my $depotpath = dirname($0);//得到执行./configure的目录
...
Getopt::Long::Configure("bundling_override");
...
# 生成指定.pro文件的Makefile文件
if ( $opt_makemakefile ) {
    makeMakefile( @ARGV );
    exit 0;
}
...
#创建release版本中需要的目录，将安装包需要的文件拷贝到安装目录下
if ( $shadow ) {
    symlink_file($depotpath."/apps",$outpath."/apps");
    mkpath("$outpath/etc");
    ...
}
# 判断选项的有效性
debugMsg("Validating options");

if ( ! -e "$depotpath/mkspecs/$opt_platform" ) {
    die "$depotpath/mkspecs/$opt_platform does not exist, check -platform parameter is correct.\n";
}
...
#以写的方式打开文件config.pri，将用户选项的选项写入config.pri文件中
open CONFIG_PRI,">$outpath/src/config.pri" or die "Can't open $outpath/src/config.pri\n";
print CONFIG_PRI "!contains( PROCESSED_PRI, config.pri ) {\nPROCESSED_PRI += config.pri\n\n";
#输入配置信息到CONFIG_PRI
...
close CONFIG_PRI;
...
# 查找.pro结尾的文件加入到@proFiles中，准备产生Makefiles
my $whichpros = ".*\\.pro\$";      #.*表示匹配0或1次的任何字符，\$表示结尾
$whichpros = "/$opt_make/$whichpros" if defined $opt_make;
#在src的各个子目录下查找.pro文件，放入数组
my @proFiles = findFiles($whichpros, "$depotpath/src");
# themes目录有.pro文件，加入数组
push(@proFiles, findFiles($whichpros, "$depotpath/etc/themes"));
```

```
# Qtopia
#从 order.cfg 读出各种库的路径，找到对应的 .pro 文件存入 @ProFiles 数组的前面
if ( $opt_qtopia ) {
    open IN, "<".$depotpath."/src/order.cfg"  #以读的方式打开文件
                    or die "Can't open ".$depotpath."/src/order.cfg\n";
    my @earlyProFiles = () ;
    while ( defined($_ = <IN>) ) {
        chomp;#去掉换行符
        my $proFile = $_."/".basename($_).".pro"; #  每行字符如 test，得到 test/test.pro
        my @tmp = grep(/\Q$proFile\E/, @proFiles);  #过滤出含 $proFile元素，组成新的数组@tmp
        if ( scalar(@tmp) ) { # 数组@tmp 的元素个数不为 0
            $proFile = $tmp[0]; # 得到第 个元素
            @proFiles = grep(!/\Q$proFile\E/, @proFiles); #删除数组中含有$proFile 的元素
            push(@earlyProFiles, $proFile);  #压入到数组的后面
        }
    }
    unshift(@proFiles, @earlyProFiles);  #加到数组的前面
    close IN;
}

# 处理 .pro文件，产生 Makefile
for my $aProFile (@proFiles) {  #遍历 @proFiles  数组中每个 .pro文件
    processProFile($aProFile); #处理 .pro 文件，生成 Makefile
}
...
exit 0;

#########################################################################
#下面是辅助函数
#########################################################################
# Find file 在一个给定路径中查找给定匹配模式的文件
#
# Parameters :
#    $re  - 搜索的文件名（含有通配符）
#    $dir - 查找文件的目录及它的子目录
#
# 返回查找到的文件
sub findFiles
{
    my ( $re, $dir ) = @_;
    my @found = ();
    find(
     sub {
         my $file = $File::Find::name;
         if ( $file =~ $re ) {
          push( @found, $file );  #将配置的文件压入到@founc 数组中，即数组末尾
         }
     }, $dir );
    return @found;
}

#省略编译 qmake
sub makeQmake
{

}

# 处理一个 .pro 文件
sub processProFile
{
    my ( $aProFile ) = @_;  # 只有一个函数参数，得到参数

    #进入源目录（ .pro 所在目录）
    my $srcDir = dirname($aProFile);  #得到文件所在的目录
```

```
    chdir $srcDir;

    # 读取 .pro 文件
    my %aProConfig;
    open PRO, "$aProFile";  #打开文件
    while (<PRO>) {
        #匹配 CONFIG+=时，存 CONFIG 值到 hash表 aProConfig 中，$1 为 CONFIG 值
        if ( /CONFIG\s*\+?=\s*(.*)/ )  {
            #先把$1 以空格分开成数组，然后把数组中每个元素($_)组成 hash表。其中数组元素作为 hash 值，hash表元素为累加计数值
            map { $aProConfig{$_}++ } split(/\s+/,$1);
        }
    }
    close PRO;

    # 得到或创建目标目录
    my $targetDir = $srcDir;
    if ( $shadow )  {
        $targetDir =~ s/^$depotpath/$outpath/;
    }
    if (! -e $targetDir) {
        mkpath($targetDir);
    }

    my $QMAKE_QTOPIA_EXTRA_ARGS = "";

    # 设置目标 Makefile 的名字
    my $makefile = "Makefile.target";
    if ( $aProFile =~ m{([^/]*-components).pro} ) { # 例如存在 src-components.pro
        #次要的 "component" pro 文件
        $makefile = "Makefile.$1";  # 例如：Makefile.src-componensts
    }
    $QMAKE_QTOPIA_EXTRA_ARGS .= " -o $makefile";

    #检查是否编译 QtopiaDesktop，设置 qmake参数
    if ( !$opt_qtopia or $aProConfig{qtopiadesktop} or $aProConfig{desktop} ) {
        $QMAKE_QTOPIA_EXTRA_ARGS .= " -spec $depotpath/mkspecs/$opt_platform";
    } else {
        $QMAKE_QTOPIA_EXTRA_ARGS .= " -spec $depotpath/mkspecs/qws/$opt_xplatform";
    }

    #运行 qmake
    chdir $targetDir;
    $ENV{'PWD'} = $targetDir;
    if ( -f $makefile ) { #  如果 $makefile 存在，删除它
        unlink $makefile;
    }
    #设置 qmake命令行
    my $qmake_cmd = "$outpath/bin/qmake $QMAKE_QTOPIA_EXTRA_ARGS $aProFile";
    if ( $isWindows ) {  #windows  操作系统中路径符是 \，因此，把 /换为 \
        $qmake_cmd =~ s,/,\\,g;
    }

    if ( !$opt_setupdepothop ) {  #执行 qmake命令，生成 Makefile.target 文件
        system($qmake_cmd) == 0 or die "Error while running qmake on $aProFile\n";
    }

    # 创建 Makefile 和 makefile 文件
    makeMakefile("$targetDir/Makefile", "$targetDir/$makefile", $aProConfig{qtopiadesktop},
                 $qmake_cmd);

}
```

用户定义 Makefile 文件，使用 make 命令再将 Makefile 重定义到 Makefile.target 文件，实际上 Makefile.target 文件才是 make 使用的真正的 makefile 文件。这样做的目和是可根据不同的用户选择定向到同一个 Makefile.target 文件，因而不需要 多次使

用 qmake 命令

```
sub makeMakefile
{
    #得到参数：Makefile 文件名、makefile 文件名、.pro 文件中 CONFIG 值、qmake 命令行
    my ( $redmake, $tmake, $qd, $qmake_cmd ) = @_;  #得到函数参数
    my $qmake_cmd_debug = $qmake_cmd;
    #将 qmake 用 qmake -d 替代，为 qmake 命令加入 debug 选项
    $qmake_cmd_debug =~ s/^([^ ]+)qmake\s+/$1qmake -d /;

    my $prefixname;
    my $prefix;
    if ( $qd ) {  #.pro 文件中 CONFIG 值存在
        $prefixname = "DPREFIX";
        $prefix = $opt_dprefix;
        $qd = 1;
    } else {
        $prefixname = "PREFIX";
        $prefix = $opt_prefix;
        $qd = 0;
    }

    if ( -f $redmake ) { #删除存在的 Makefile
        unlink $redmake or die "I can't unlink $redmake: $!\n";
    }

    #我们想有一个定义目标的最小设置，即使 Makefile.target 不含有它们。因为在 Makefile.target 被正确编译时，它将包括这个最小设置，最好不需要再配置，因为 Makefile.target 将被再编译
    my @targets = qw( all distclean clean install install_target uninstall packages lupdate sdk devsdk );
#定义数组@targets 中各元素的字符串
    if ( open TMAKE, "<".$tmake ) {  #tmake 在 Qtopia 2.1.1 中已可不使用了
        ......
        close TMAKE;
    }

    #变量$contents 存入将写入 Makefile 中的内容
    my $contents = "";
    if ( $isWindows ) { #是 Windows 操作系统
        ......
    } else { #非 Windows 操作系统
        #加入 PREFIX?=/home/test/qt_soft/qtopia-free-2.1.1/image/opt/Qtopia
        $contents .= "$prefixname?=$prefix\n\n";

        #用 for 循环对不同的编译方式，将用户定义的编译规则写入$contents变量中，在 Makefile 中所见到的两个在 define 与 endef 之间的内容块，即由 for 的两次循环产生这两个内容块的内容除了个别参数不一样外，其他基本一致，因而使用循环来产生
        for (
        # BUILD 编译方式及编译命令
         ["BUILD","\$(MAKE) -f Makefile.target \$\@ INSTALL_ROOT=\$($prefixname)"],
        # QMAKE_BUILD 编译方式及编译命令
         ["QMAKE_BUILD",
          "\$(MAKE) -f Makefile.target -n \$\@ INSTALL_ROOT=\$($prefixname)>/dev/null 2>&1 ".
           "|| exec \$(MAKE) regenerate; ".                    # '.' 表示字符串连接
              "\$(MAKE) -f Makefile.target \$\@ INSTALL_ROOT=\$($prefixname)"] )
        {
            my ( $rule, $command ) = @$_; #将编译方式存入$rule，编译命令存入$command
            my $shadow_command = $command;
            #在字符串$shadow_command 中，将所有的 Makefile.target 替换为 Makefile
            $shadow_command =~ s,Makefile\.target,Makefile,g;
            #以下是加入 Makefile 的内容，直到 END 行为止，是为 Makefile 编写的规则
            $contents .= <<END;
#将直到 END部分给字符串处理后加入 Makefile中，具体内容参看 Makefile
define $rule  #在 Makefile 中内容为：define BUILD

            END
        }
        #在$contents 中加入其他规则
```

```
        for my $target ( @targets ) {
            $contents .= "$target:\n\t\$(BUILD)\n";
        }
        ......
    }

    #将$contents写入$redmake文件，即Makefile文件
    open REDMAKE, ">".$redmake or die "Unable to write to $redmake: $!\n";
       print REDMAKE $contents;
    close REDMAKE;

    my $tmp_redmake = $redmake;
    my $tmp_srcmake = "$outpath/src/Makefile";
    ......
    if ( $tmp_redmake eq $tmp_srcmake ) {  #如果Makefile文件已存在
        # 写顶层Makefile (从src/Makefile拷贝规则)
        ......
    }
}
```

8.6 Qtopia 的编译过程

Qtopia 的编译包括库、应用程序及插件等，由于编译时应用程序与库及插件之间可能存在依赖关系，因此编译有先后顺序，这种编译顺序如何从项目文件传递到 Makefile 中？

在 global.pri 项目文件中定义了整个编译条目的组成，整个编译条目由系统变量 ALL_PROJECTS 定义，它按编译的顺序将编译条目分成库、应用程序、插件、服务器及主题条目。在执行 qmake 命令时，qmake 分别按照各个子条目中的目标次序在 Makefile.target 中生成对应的语句。同时将项目文件中提到的特殊依赖关系，转换为 Makefile 中的依赖关系。

在 global.pri 中，ALL_PROJECTS 定义的定义语句列出如下：

```
ALL_PROJECTS=$${LIBRARY_PROJECTS} $${APP_PROJECTS} $${PLUGIN_PROJECTS} $${SERVER} $${THEME_PROJECTS}
```

在 src.pro 中，编译 Qtopia 的语句列出如下：

```
buildQtopia {
    include($${QTOPIA_DEPOT_PATH}/src/projects.pri)
    CONFIG+=ordered
    SUBDIRS+=$${LIBRARY_PROJECTS} src-components.pro
}
```

在 /src/projects.pri 中，核心应用程序及库的配置放在 general.pri 项目文件中，projects.pri 中与 general.pri 相关的语句列出如下：

```
include($${QTOPIA_DEPOT_PATH}/src/general.pri)
```

8.6.1 编译库

ALL_PROJECTS 系统变量中的第 1 个组成系统变量是 LIBRARY_PROJECTS，它表示先编译库。在 projects.pri 中，LIBRARY_PROJECTS 由核心库变量与普通库变量组成，LIBRARY_PROJECTS 的定义列出如下：

```
LIBRARY_PROJECTS=$$CORE_LIBRARY_PROJECTS $$LIBRARY_PROJECTS
```

从上面的语句可以看出，最先进行 CORE_LIBRARY_PROJECTS 系统变量定义库的编译，然后进

行 LIBRARY_PROJECTS 系统变量所定义的库的编译。

从顶层的 Makefile.target 中可以看出 LIBRARY_PROJECTS 系统变量按照目录加入次序进行编译，同时有依赖关系的还需要检查依赖关系。 src.pro 中的模板为 TEMPLATE=subdirs ，系统变量 SUBDIRS 的定义列出如下：

```
SUBDIRS+=$${LIBRARY_PROJECTS} src-components.pro
```

执行 qmake 命令后，项目文件中的系统变量 SUBDIRS 转换为在 Makefile.target 中下面的语句：

```
SUBTARGETS =  \
        sub-qt \                    #加入“sub-”前缀
        sub-libraries-qtopia \      #文件路径的“/”符号变成了“-”连接号
        sub-libraries-qtopia1 \
        sub-libraries-qtopia2 \
        sub-3rdparty-plugins-obex-openobex \
        sub-libraries-qtopiapim \
        sub-libraries-mediaplayer \
        sub-libraries-qtopiacalc \
        sub-libraries-qtopiapim1 \
        sub-3rdparty-libraries-freetype \
        sub-3rdparty-libraries-gsm \
        sub-libraries-handwriting \
        sub-src-components_pro
```

在使用 make 命令编译时，make 会先检查依次 SUBTARGETS 对应的条目之间的依赖关系，并按依赖关系编译各个条目。没有特殊的依赖关系，则按 SUBTARGETS 对应条目的次序进行编译。例如，对于 sub-libraries-qtopia 来说，它进入的操作是：

```
sub-libraries-qtopia: libraries/qtopia/$(MAKEFILE) FORCE
    cd libraries/qtopia && $(MAKE) -f $(MAKEFILE)
```

sub-libraries-qtopia 又依赖于 libraries/qtopia/$(MAKEFILE)，因此它会检查 libraries/qtopia/$(MAKEFILE)是否更新。若 libraries/qtopia/$(MAKEFILE)有文件更新变化（如被删除），则执行 libraries/qtopia/$(MAKEFILE) 所在语句，这个语句列出如下：

```
libraries/qtopia/$(MAKEFILE):
    @$(CHK_DIR_EXISTS) "libraries/qtopia" || $(MKDIR) "libraries/qtopia"
    cd libraries/qtopia && $(QMAKE) qtopia.pro -spec
     /home/test/qt_soft/qtopia-free-2.1.1/mkspecs/qws/linux-generic-g++ -o $(MAKEFILE)
```

qmake 还根据在项目文件中的依赖关系在 Make.target 中产生依赖关系描述语句。下面以 global.pri 项目文件代码为例说明库的依赖关系如何从项目文件转换到 Makefile 文件中。

在 global.pri 项目文件中部分库依赖关系描述如下：

```
!contains(TEMPLATE,subdirs) {
    qtopiaapp:CONFIG+=qtopia  #配置qtopia
    qtopia:CONFIG+=qtopialib qtopiainc #配置的qtopia依赖于这两项，需要配置这两项

    #基本库
    qtopiadesktop {
        ......
    } else {
       pimlib:LIBS+=-lqpepim1 -lqpepim
       qtopialib { #这个库需要连接下面这些库
         QTOPIA_PHONE:LIBS+=-lqtopiaphone
         LIBS+=-lqtopia2 -lqtopia -lqpe
       }
       qssaudio:DEFINES+=QSSAUDIO
    }
```

运行 qmake 命令后，在 Makefile.target 中基本库的依赖关系描述语句的一部分列出如下：

```
sub-libraries-qtopia: sub-qt
sub-libraries-qtopia1: sub-libraries-qtopia
sub-libraries-qtopia2: sub-libraries-qtopia1
```

8.6.2 相互独立的应用程序及插件的编译

相互独立的应用程序及插件的编译在项目文件 src-components.pro 中描述，它是一个模板为 TEMPLATE=subdirs 的项目文件，系统变量 SUBDIRS 的定义列出如下：

```
SUBDIRS+=$${APP_PROJECTS} $${PLUGIN_PROJECTS} $${SERVER}
```

gerneral.pri 项目文件给出应用程序编译条目的定义，编译条目的部分列出如下：

```
APP_PROJECTS+=\
    applications/qss \
    ......\
    tools/quicklauncher \
    tools/qcop \
    tools/symlinker
```

执行 qmake 命令后，项目文件中的系统变量 SUBDIRS 转换为 Makefile.src-components 中下面的语句：

```
SUBTARGETS =   \
          sub-applications-addressbook \
          sub-applications-calculator \
          ......\
          sub-plugins-codecs-wavrecord \
          sub-server \
          sub-  -etc-themes
```

8.6.3 文件的安装

文件的安装包括编译生成的应用程序和库文件，以及用户需要拷贝到目标设备中的文件。

（1）安装 SUBDIRS 中条目指定的目录

对于系统变量 SUBDIRS 所指的目录，qmake 会生成类似于以下的语句，使其调用系统变量 SUBDIRS 中条目所指的 Makefile.target 来安装。下面列出 SUBDIRS 所指的一个目录对应的安装语句（在 src/Makefile.target 中）：

```
install_subdirs: qmake_all FORCE
      (
        [ -d qt ] && cd qt ;      #如果 qt 是目录，进入 qt
        #-n 表示打印出将执行的命令，但不执行。为了不在终端上显示，重定向输出到 null 设备
        if $(MAKE) -n -f $(MAKEFILE) install >/dev/null 2>&1;
        then $(MAKE) -f $(MAKEFILE) install;  #在 qt 下运行 make install 执行安装
         fi;
    )
```

（2）安装源代码中文件

有的应用程序或资源在源代码中已存在，需要在安装时拷贝到安装目录下。这种安装拷贝需要在项目文件中写出安装的描述语句，如写出相应的查找安装命令及将安装条目加入到 INSTALL 系统变

量中。qmake命令在运行时，根据 INSTALL 系统变量中的条目，将这些语句转换成 Makefile.target 文件中的安装语句。

在项目文件中安装条目中的 command 为安装命令字符，files 为需要安装的文件，path 表示安装路径。

下面分别列出安装条目中的 command 与 files 从.pro 项目文件中安装语句转换为 Makefile 中安装语句。

在 src.pro 中 nonpropackagesinstall 的安装条目列出如下：

```
#从 configure 脚本中可查到变量 COMMAND_HEADER=\@
nonpropackagesinstall.commands=$${COMMAND_HEADER}\  #$${COMMAND_HEADER}=\@
for i in \   #在目录下查找*.control 的文件
    `find $${QTOPIA_DEPOT_PATH}/services -name '*.control'` \
    `find $${QTOPIA_DEPOT_PATH}/sounds -name '*.control'` \
    `find $${QTOPIA_DEPOT_PATH}/misc -name '*.control'` \
;do \
    echo Installing files listed in \$$i;\  #用 echo 打印信息到终端窗口中
    #如果$$i 中含有“Files:”前缀，将它用空格替换，然后赋给 FILELIST
    FILELIST=`grep "^Files:" \$$i | sed -e 's/^Files://'`;\
    [ -z "\$$FILELIST" ] || for j in \$$FILELIST; do \
     dir=`dirname \$$j`;\  #得到$$j 的目录名
     mkdir -p "$(INSTALL_ROOT)/\$$dir";\  #如果$$dir 目录不存在，则创建；如果存在，不执行任何操作
     eval install -c "$$(QPEDIR)/\$$j" "$(INSTALL_ROOT)/\$$dir";\  #安装拷贝到目录$$dir
    done;\
done;\

nonpropackagesinstall.CONFIG+=no_path
buildQtopia:INSTALLS+=nonpropackagesinstall   #加入到 INSTALL 系统变量中，表示是安装拷贝操作
```

src.pro 中 nonpropackagesinstall 的安装条目中的 command 字符串被完整地移到 Makefile.target 中，nonpropackagesinstall 的安装语句列出如下：

```
#在 INSTALLS 系统变量中的条目前加上“install_”前缀
install_nonpropackagesinstall: all
    #下面是 nonpropackagesinstall.commands 中的内容
     @ for i in `find /home/test/qt_soft/qtopia-free-2.1.1/services -name '*.control'`
       `find /home/test/qt_soft/qtopia-free-2.1.1/sounds -name '*.control'`
        `find /home/test/qt_soft/qtopia-free-2.1.1/misc -name '*.control'` ;

    do echo Installing files listed in $$i;
        FILELIST=`grep "^Files:" $$i | sed -e 's/^Files://'`;
       [ -z "$$FILELIST" ] || for j in $$FILELIST;

          do dir=`dirname $$j`;
            mkdir -p "$(INSTALL_ROOT)/$$dir";
            eval install -c "/home/test/qt_soft/qtopia-free-2.1.1/$$j"
                        "$(INSTALL_ROOT)/$$dir";
          done;
     done;
```

在项目文件中安装条目中的 files 表示需要安装的文件，项目文件 src.pro 中的 sdk_qmake 安装条目中的 files 和 path，列出如下：

```
sdk_qmake.files=\    #需要安装的文件
     $$(QPEDIR)/bin/qmake\
     $$(QPEDIR)/bin/nct_lupdate\
     $$(QPEDIR)/bin/installpic\
     $$(QPEDIR)/bin/mkipks
    sdk_qmake.path=/bin  #安装的目录
    sdk_qmake.CONFIG=no_default_install
INSTALLS+=sdk_qmake
```

执行 qmake 命令后，sdk_qmake 安装条目被转换为 Makefile.target 中的安装语句，列出如下：

```
install_sdk_qmake: all
    #检查/bin 目录是否存在，若不存在，创建一个；否则不执行任何操作
     @$(CHK_DIR_EXISTS) "$(INSTALL_ROOT)/bin/" || $(MKDIR) "$(INSTALL_ROOT)/bin/"
    #安装拷贝源文件到目的路径
     -$(INSTALL_FILE) "../bin/qmake" "$(INSTALL_ROOT)/bin/"  #INSTALL_FILE= install -c
     -$(INSTALL_FILE) "../bin/nct_lupdate" "$(INSTALL_ROOT)/bin/"
     -$(INSTALL_FILE) "../bin/installpic" "$(INSTALL_ROOT)/bin/"
     -$(INSTALL_FILE) "../bin/mkipks" "$(INSTALL_ROOT)/bin/"
```

8.6.4 打包分发

Qtopia 设备的软件包应该以 IPKG(Itsy Package Management System)格式分发，Linux/X11 的包应以 RPM（Red Hat Package Manager）格式分发，这两种格式的包都可以用 Qtopia SDK 的 mkipks 工具程序来生成。应用程序采用 .control 文件来描述应用程序相关信息，一个.control 样例文件 qpe-clock.control 列出如下：

```
#$QPEDIR 的相对路径下的文件列表，可使用通配符
Files: bin/clock apps/Applications/clock.desktop services/TimeMonitor/clock
Priority: optional    #优先级，正常是 optional
#包类型，如 qpe/applications，qpe/games，qpe/settings，qpe/system，kernel 等
Section: qpe/applications
Maintainer: Trolltech (www.trolltech.com)   #包的维护者
Architecture: $CPU_ARCH
Arch: $DEVICE_ARCH
Version: $QPE_VERSION-12
Depends: qpe-libqtopia  #依赖的包的列表
Description: Clock and stop-watch  #非常简短的描述
#多行描述
 A simple clock and stop-watch for the Qtopia environment.
```

对于 IPKG 格式，还可以用表 8-20 所示的后缀文件代替.control 后缀的文件。

表 8-20 后缀文件

后　缀	说　明
.preinst	在文件被移到安装位置之前执行
.postinst	在文件被移到安装位置之后执行，通常如下使用。 （1）/opt/Qtopia/bin/qcop QPE/TaskBar "reloadInputMethods()"：如果软件是一个输入法插件，再次安装输入法。 （2）/opt/Qtopia/bin/qcop QPE/TaskBar "reloadApplets()"：如果软件是一个 taskbar applet ，再次装载 applets
.prerm	在文件卸载之前执行
.postrm	在文件卸载之后执行，通常如下使用。 （1）/opt/Qtopia/bin/qcop QPE/TaskBar "reloadInputMethods()"：如果软件是一个输入法插件，再次安装输入法。 （2）/opt/Qtopia/bin/qcop QPE/TaskBar "reloadApplets()"：如果软件是一个 taskbar applet ，再次装载 applets

文件 src/ipk_groups.pri 定义分组打包的系统变量，src/gloabal.pri 定义如何打包。一般情况下是一个项目文件打一个 ipkg 包，即一个应用程序或库打一个包。有时，多个项目被打入一个 ipkg 包。src/gloabal.pri 定义的打包方法被加到应用程序或库的 Makefile.target 中。

当需要将单个应用程序打包时，进入这个应用程序的 Makefile 所在目录，运行命令 make packages。Makefile.target 会调用工具 mkipks （用于嵌入设备）或 tar（用于 X11）完成打包操作，打包完成后生成*.ipk 或 tar 结尾的文件。在嵌入设备上安装时，使用命令 ipkg 即可将*.ipk 包解开，并安装到设备上。ipkg 命令的用法与 tar 命令有些类似，具体可使用 ipkg --help 来查看 ipkg 的帮助信息。

当需要将 Qtopia 中所有的应用程序或库等打包时，在顶层运行 make packages 命令，这个命令会到各个应用程序或库的目录下运行 make packages 命令来将各个应用程序或库打包。顶层运行 make packages 命令执行的语句（在 src/Makefile.target 中）列出如下：

```
packages: nonpropackages
    @ [ -z "qt libraries/qtopia ..... libraries/handwriting src-components.pro;
   do if [ -e $$i/Makefile ];   #如果$$i 目录下的 Makefile 存在
        #进入$$i 目录，运行 make packages，$@表示为运行 make 的参数。如果运行不成功，退出并打印错误信息。$$?表
示打印错误信息
        then ( cd $$i; $(MAKE) $@ || exit $$?; )
     fi;
  done;

   #将$$i 目录中的*.pro 文件名的.pro 以空格替换，这样形成类似 Makefile.src-components 的字符串
   COMPONENT_MKFILE=Makefile.`echo $$i | sed -e 's,\.pro,,'`;
   if [ -e "$$COMPONENT_MKFILE" ];
     then $(MAKE) -f $$COMPONENT_MKFILE $@ || exit $$?;
   fi;
```

在 src/gloabal.pri 中，打包方法用系统变量 unix_mkipk_deps 和 unix_mkipk 描述。它被加入到 QMAKE_EXTRA_UNIX_TARGETS 附加编译目标中，相关语句（在 src/gloabal.pri 中）列出如下：

```
QMAKE_EXTRA_UNIX_TARGETS+=unix_mkipk unix_mkipk_deps
        packages.depends=unix_mkipk
```

gloabal.pri 中的系统变量 unix_mkipk 的 command 条目含打包的具体方法，它根据平台来决定使用 tar 还是 mkipks 工具来打包。依据平台将一些变量替换后， command 条目的内容几乎不变地移到 Makefile.target 中。

打包的过程是首先将从源目录可执行的应用程序安装拷贝到.packaging-working-dir/ipk/qpe-clock/ 中，并在其中设置包描述文件。然后从源目录拷贝特殊描述文件＊ .preinst 等到这个目录中并将这个目录打包，并且删除临时文件及目录。

Makefile.target 中与打包相关的代码（在 src/applications/clock/Makefile.target 中）列出如下：

```
unix_mkipk:
    @ ......  #判断包是否存在，若存在，则退出
   [ -z "applications/clock" ]
|| for i in applications/clock;  #这里只有一项，i 为 applications/clock
    #建立包所有的目录，安装拷贝编译生成的应用程序
   do make -C /home/test/qt_soft/qtopia-free-2.1.1/src/$$i   #进入$$i 目录
        -f Makefile.target unix_mkipk_deps
             INSTALL_ROOT=/home/test/qt_soft/qtopia-free-2.1.1/
                  .packaging-working-dir/ipk/qpe-clock/data/opt/Qtopia
     || exit $$?; #执行不成功时，退出并打印出错信息
  done;

  #设置包描述文件 control，拷贝特殊的描述文件*.preinst 等，打包并删除临时文件
  make_ipk_file() {
     cd /home/test/qt_soft/qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock/data;

     if [ ! -z "$$1" ];  #$$1 为函数第 1 个参数，如 en_US
        then lang="$$1";
      suffix="_i18n_$$lang";
      deps="qpe-clock";
      if [ "`echo qpe-clock | cut -c -4`" = "qpe-" -a "qpe-clock" != "qpe-taskbar" ];
        then deps="$$deps qpe-taskbar_i18n_$$lang";
      fi;

      path="opt/Qtopia/i18n/$$lang";
      #如果$$path 不是目录，返回
```

```
    if [ ! -d "/home/test/qt_soft/qtopia-free-2.1.1/
              .packaging-working-dir/ipk/qpe-clock/data/$$path" ];
       then return;
    fi;
    else suffix=;
       #将 DEPS 文件中依赖关系内容导入到 deps 变量中
       deps=`cat home/test/qt_soft/qtopia-free-2.1.1/
              .packaging-working-dir/ipk/qpe-clock/control/DEPS`;
       deps=`echo $$deps | sort -u`; #排序
       path="opt/Qtopia";
    fi;

    #设置 control 文件的描述语句
    echo "Package: qpe-clock$$suffix" > /home/test/qt_soft/qtopia-free-2.1.1/
                        .packaging-working-dir/ipk/qpe-clock/control/control;
     ......
      #拷贝安装 qpe-clock.preinst 这样的文件
     for i in preinst prerm postinst postrm;
        do export FILE_LOC=/home/test/qt_soft/qtopia-free-2.1.1/
                      src/applications/clock/qpe-clock.$$i;

          if [ -e $$FILE_LOC ];  #拷贝安装$$FILE_LOC 文件
            then install -c $$FILE_LOC /home/test/qt_soft/qtopia-free-2.1.1/
                  .packaging-working-dir/ipk/qpe-clock/$$i;
          fi;
        done;

        cd /home/test/qt_soft/qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock;
        echo "2.0" > debian-binary;

        cd /home/test/qt_soft/qtopia-free-2.1.1/
                    .packaging-working-dir/ipk/qpe-clock/control;
         #因为选择的平台是 X11，因此，使用工具 tar 将 control 目录打包
        tar --remove-files -cz --owner=0 --group=0 -f  /home/test/qt_soft/
            qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock/control.tar.gz ./*;

       #将 data 目录打包
       cd /home/test/qt_soft/qtopia-free-2.1.1/
                      .packaging-working-dir/ipk/qpe-clock/data;
       tar --remove-files -cz --owner=0 --group=0 -f /home/test/qt_soft/
            qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock/data.tar.gz $$path;

       #创建或得到 ipkg 目录
      if [ ! -e /home/test/qt_soft/qtopia-free-2.1.1/ipkg ];
         then mkdir -p /home/test/qt_soft/qtopia-free-2.1.1/ipkg;
      fi;

     #将安装拷贝 control 包描述文件的 qpe-clock 目录打包
     cd /home/test/qt_soft/qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock;
     tar --remove-files -cz --owner=0 --group=0 -f /home/test/qt_soft/qtopia-free-2.1.1/
               ipkg/qpe-clock_2.1.1_x86$$suffix.ipk
                 ./debian-binary ./control.tar.gz ./data.tar.gz `

     #打印 preinst 等
    for i in preinst prerm postinst postrm;
      do if [ -e $$i ];
      then echo $$i;
      fi;
  done`;
};

[ -z "en_US" ]
|| for lang in en_US;
do make_ipk_file $$lang;
done;
```

```
    #删除临时目录
    { -d /home/test/qt_soft/qtopia-free-2.1.1/
                .packaging-working-dir/ipk/qpe-clock/data/opt/Qtopia/i18n }
     && rm -r /home/test/qt_soft/qtopia-free-2.1.1/
                .packaging-working-dir/ipk/qpe-clock/data/opt/Qtopia/i18n;
     make_ipk_file;
    rm -rf /home/test/qt_soft/qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock

#建立包所有的目录，通过它依赖的 install 目标来安装拷贝编译生成的应用程序
unix_mkipk_deps: install
     @ mkdir -p /home/test/qt_soft/qtopia-free-2.1.1/
                        .packaging-working-dir/ipk/qpe-clock/data; #创建目录
     mkdir -p /home/test/qt_soft/qtopia-free-2.1.1/
                 .packaging-working-dir/ipk/qpe-clock/control; #创建目录
    #将依赖的库名经字符替换后以追加的方式写 DEPS 文件名
    echo "qpe-libqtopia2" | sed 's/\([^(]\)/,\1/g' >> /home/test/qt_soft/
               qtopia-free-2.1.1/.packaging-working-dir/ipk/qpe-clock/control/DEPS
#执行 make packages 操作的入口处
packages: unix_mkipk
```

8.7 生成 img 文件并烧录

嵌入式 Linux 系统的烧录包括 3 个层次，即引导装载程序、Linux 内核和用户界面。引导装载程序最先执行，在嵌入设备上通常没有 BIOS，它使用一些专用程序和引导程序。专用程序能直接读写设备上的闪存，并将引导装载程序烧录在闪存的指定位置上。例如，JFlash 工具在主机上执行后可通过嵌入设备的 JTAG 接口使用并行端口访问目标设备的闪存。 JFlash 工具初始化 DRAM 并运行设备的个串行或 USB 或以太网端口的控制程序，建立与主机上对应的应用程序的通信。这样，主机应用程序即连接设备并把引导装载程序下载到设备中，常用的开放源代码的引导装载程序有 Blob、Redboot 和 Bootldr。

引导装载程序的功能有执行加电自检（POST）、初始化 CPU 速度、初始化内存及下载用端口、设置堆栈指针、设置参数结构并设置在闪存中的位置标记，以及跳转到内核的开始地址。

内核编译后生成 zImage 文件，这个文件被烧录到闪存的指定位置。它不需要文件系统，编译生成的模块放在 jffs2 文件系统的 /lib 下。

用户界面应用程序等不需要修改的软件可被放入只读压缩的 cramfs 文件系统中，常用方法是将这些软件拷贝到一个目录，然后使用 mkcramfs 命令生成.img 文件。例如，Qtopia 的编译安装后的软件放在$QPEDIR/image下，可以使用下面的命令将其打包为 .img 包：

```
cd $QPEDIR
mkcramfs  image  qpe.img
```

将常用目录（如/usr、/sbin、/bin 和/etc 等）及其文件打包成 jffs2 文件系统，方法一样。首先将需要的文件拷贝到这些目录下，然后使用 mkfs.jffs2 命令打包生成。例如，常用目录放在/test/root 下，如下使用命令 mkfs.jffs2 打包生成 img 文件：

```
mkfs.jffs2  -d /test/root -o root.img -e 128 -f
```

上述命令中的-e 128 表示闪存的擦除块大小为 128 KB，-f 表示将所有文件时间改变为 0，这是为了测试及使用的方便。

需要烧录的软件打成 *.img 包后可以开始烧录，烧录的方法有两种，一是使用 PC 机上的应用程序，JFlash 工具是使用较多的工具，这种方法一般被开发及调试人员使用；二是直接使用烧录器将软件烧录到闪存中，这种方法适用于批量生产。

烧录方法的具体细节与具体的处理器芯片型号有关，由处理器芯片生厂商提供。

主要参考文献

1 The XFree86 Project Inc. Documentation for XFree86® version 4.5.0. http://www.xfree86.org/current/

2 Daniel Manrique. X Window System Architecture Overview HOWTO. http://www.faqs.org/docs/Linux-HOWTO/XWindow-Overview-HOWTO.html

3 KDE Documentation. http://www.kde.org/documentation/

4 KDE 3 Architecture. http://developer.kde.org/documentation/library/kdeqt/kde3arch/index.html

5 Qt/embedded 3.4 源代码所带文档. http://www.trolltech.com

6 Qtopia2.1.1 源代码所带文档. http://www.trolltech.com